A friendly introduction to web programming

the little ASP.NET Core book

Nate Barbettini

Table of Contents

The Little ASP.NET Core Book

by Nate Barbettini

ISBN: 978-1-387-75615-5

Introduction

Thanks for picking up The Little ASP.NET Core Book! I wrote this short book to help developers and people interested in web programming learn about ASP.NET Core, a new framework for building web applications and APIs.

The Little ASP.NET Core Book is structured as a tutorial. You'll build an application from start to finish and learn:

- The basics of the MVC (Model-View-Controller) pattern
- How front-end code (HTML, CSS, JavaScript) works together with back-end code
- What dependency injection is and why it's useful
- How to read and write data to a database
- How to add log-in, registration, and security
- How to deploy the application to the web

Don't worry, you don't need to know anything about ASP.NET Core (or any of the above) to get started.

Before you begin

The code for the finished version of the application you'll build is available on GitHub:

https://www.github.com/nbarbettini/little-aspnetcore-todo

Feel free to download it if you want to see the finished product, or compare as you write your own code.

The book itself is updated frequently with bug fixes and new content. If you're reading a PDF, e-book, or print version, check the official website (littleasp.net/book) to see if there's an

updated version available. The very last page of the book contains version information and a changelog.

Reading in your own language

Thanks to some fantastic multilingual contributors, the Little ASP.NET Core Book has been translated into other languages:

- **ASP.NET Core El Kitabı** (Turkish)
- **简明 ASP.NET Core 手册** (Chinese)

Who this book is for

If you're new to programming, this book will introduce you to the patterns and concepts used to build modern web applications. You'll learn how to build a web app (and how the big pieces fit together) by building something from scratch! While this little book won't be able to cover absolutely everything you need to know about programming, it'll give you a starting point so you can learn more advanced topics.

If you already code in a backend language like Node, Python, Ruby, Go, or Java, you'll notice a lot of familiar ideas like MVC, view templates, and dependency injection. The code will be in C#, but it won't look too different from what you already know.

If you're an ASP.NET MVC developer, you'll feel right at home! ASP.NET Core adds some new tools and reuses (and simplifies) the things you already know. I'll point out some of the differences below.

No matter what your previous experience with web programming, this book will teach you everything you need to create a simple and useful web application in ASP.NET Core. You'll learn how to build functionality using backend and frontend code, how to interact with a database, and how to deploy the app to the world.

What is ASP.NET Core?

ASP.NET Core is a web framework created by Microsoft for building web applications, APIs, and microservices. It uses common patterns like MVC (Model-View-Controller), dependency injection, and a request pipeline comprised of middleware. It's open-source under the Apache 2.0 license, which means the source code is freely available, and the community is encouraged to contribute bug fixes and new features.

ASP.NET Core runs on top of Microsoft's .NET runtime, similar to the Java Virtual Machine (JVM) or the Ruby interpreter. You can write ASP.NET Core applications in a number of languages

(C#, Visual Basic, F#). C# is the most popular choice, and it's what I'll use in this book. You can build and run ASP.NET Core applications on Windows, Mac, and Linux.

Why do we need another web framework?

There are a lot of great web frameworks to choose from already: Node/Express, Spring, Ruby on Rails, Django, Laravel, and many more. What advantages does ASP.NET Core have?

- **Speed.** ASP.NET Core is fast. Because .NET code is compiled, it executes much faster than code in interpreted languages like JavaScript or Ruby. ASP.NET Core is also optimized for multithreading and asynchronous tasks. It's common to see a 5-10x speed improvement over code written in Node.js.

- **Ecosystem.** ASP.NET Core may be new, but .NET has been around for a long time. There are thousands of packages available on NuGet (the .NET package manager; think npm, Ruby gems, or Maven). There are already packages available for JSON deserialization, database connectors, PDF generation, or almost anything else you can think of.

- **Security.** The team at Microsoft takes security seriously, and ASP.NET Core is built to be secure from the ground up. It handles things like sanitizing input data and preventing cross-site request forgery (CSRF) attacks, so you don't have to. You also get the benefit of static typing with the .NET compiler, which is like having a very paranoid linter turned on at all times. This makes it harder to do something you didn't intend with a variable or chunk of data.

.NET Core and .NET Standard

Throughout this book, you'll be learning about ASP.NET Core (the web framework). I'll occasionally mention the .NET runtime, the supporting library that runs .NET code. If this already sounds like Greek to you, just skip to the next chapter!

You may also hear about .NET Core and .NET Standard. The naming gets confusing, so here's a simple explanation:

.NET Standard is a platform-agnostic interface that defines features and APIs. It's important to note that .NET Standard doesn't represent any actual code or functionality, just the API definition. There are different "versions" or levels of .NET Standard that reflect how many APIs are available (or how wide

the API surface area is). For example, .NET Standard 2.0 has more APIs available than .NET Standard 1.5, which has more APIs than .NET Standard 1.0.

.NET Core is the .NET runtime that can be installed on Windows, Mac, or Linux. It implements the APIs defined in the .NET Standard interface with the appropriate platform-specific code on each operating system. This is what you'll install on your own machine to build and run ASP.NET Core applications.

And just for good measure, **.NET Framework** is a different implementation of .NET Standard that is Windows-only. This was the only .NET runtime until .NET Core came along and brought .NET to Mac and Linux. ASP.NET Core can also run on Windows-only .NET Framework, but I won't touch on this too much.

If you're confused by all this naming, no worries! We'll get to some real code in a bit.

A note to ASP.NET 4 developers

If you haven't used a previous version of ASP.NET, skip ahead to the next chapter.

ASP.NET Core is a complete ground-up rewrite of ASP.NET, with a focus on modernizing the framework and finally decoupling it from System.Web, IIS, and Windows. If you

remember all the OWIN/Katana stuff from ASP.NET 4, you're already halfway there: the Katana project became ASP.NET 5 which was ultimately renamed to ASP.NET Core.

Because of the Katana legacy, the `Startup` class is front and center, and there's no more `Application_Start` or `Global.asax`. The entire pipeline is driven by middleware, and there's no longer a split between MVC and Web API: controllers can simply return views, status codes, or data. Dependency injection comes baked in, so you don't need to install and configure a container like StructureMap or Ninject if you don't want to. And the entire framework has been optimized for speed and runtime efficiency.

Alright, enough introduction. Let's dive in to ASP.NET Core!

Your first application

Ready to build your first web app with ASP.NET Core? You'll need to gather a few things first:

Your favorite code editor. You can use Atom, Sublime, Notepad, or whatever editor you prefer writing code in. If you don't have a favorite, give Visual Studio Code a try. It's a free, cross-platform code editor that has rich support for writing C#, JavaScript, HTML, and more. Just search for "download visual studio code" and follow the instructions.

If you're on Windows, you can also use Visual Studio to build ASP.NET Core applications. You'll need Visual Studio 2017 version 15.3 or later (the free Community Edition is fine). Visual Studio has great code completion and refactoring support for C#, although Visual Studio Code is close behind.

The .NET Core SDK. Regardless of the editor or platform you're using, you'll need to install the .NET Core SDK, which includes the runtime, base libraries, and command line tools you need for building ASP.NET Core applications. The SDK can be installed on Windows, Mac, or Linux.

Once you've decided on an editor, you'll need to get the SDK.

Get the SDK

Search for "download .net core" and follow the instructions on Microsoft's download page to get the .NET Core SDK. After the SDK has finished installing, open up the Terminal (or PowerShell on Windows) and use the `dotnet` command line tool (also called a **CLI**) to make sure everything is working:

```
dotnet --version

2.1.104
```

You can get more information about your platform with the `--info` flag:

```
dotnet --info

.NET Command Line Tools (2.1.104)

Product Information:
 Version:            2.1.104
 Commit SHA-1 hash:  48ec687460

Runtime Environment:
 OS Name:     Mac OS X
 OS Version:  10.13

(more details...)
```

If you see output like the above, you're ready to go!

Hello World in C#

Before you dive into ASP.NET Core, try creating and running a simple C# application.

You can do this all from the command line. First, open up the Terminal (or PowerShell on Windows). Navigate to the location you want to store your projects, such as your Documents directory:

```
cd Documents
```

Use the `dotnet` command to create a new project:

```
dotnet new console -o CsharpHelloWorld
```

The `dotnet new` command creates a new .NET project in C# by default. The `console` parameter selects a template for a console application (a program that outputs text to the screen). The `-o CsharpHelloWorld` parameter tells `dotnet new` to create a new directory called `CsharpHelloWorld` for all the project files. Move into this new directory:

```
cd CsharpHelloWorld
```

`dotnet new console` creates a basic C# program that writes the text `Hello World!` to the screen. The program is comprised of two files: a project file (with a `.csproj` extension) and a C# code file (with a `.cs` extension). If you open the former in a text or code editor, you'll see this:

CsharpHelloWorld.csproj

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

The project file is XML-based and defines some metadata about the project. Later, when you reference other packages, those will be listed here (similar to a `package.json` file for npm). You won't have to edit this file by hand very often.

Program.cs

```
using System;

namespace CsharpHelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

`static void Main` is the entry point method of a C# program, and by convention it's placed in a class (a type of code structure or module) called `Program`. The `using` statement at the top imports the built-in `System` classes from .NET and makes them available to the code in your class.

From inside the project directory, use `dotnet run` to run the program. You'll see the output written to the console after the code compiles:

```
dotnet run

Hello World!
```

That's all it takes to scaffold and run a .NET program! Next, you'll do the same thing for an ASP.NET Core application.

Create an ASP.NET Core project

If you're still in the directory you created for the Hello World sample, move back up to your Documents or home directory:

```
cd ..
```

Next, create a new directory to store your entire project, and move into it:

```
mkdir AspNetCoreTodo
cd AspNetCoreTodo
```

Next, create a new project with `dotnet new`, this time with some extra options:

```
dotnet new mvc --auth Individual -o AspNetCoreTodo
cd AspNetCoreTodo
```

This creates a new project from the `mvc` template, and adds some additional authentication and security bits to the project. (I'll cover security in the *Security and identity* chapter.)

> You might be wondering why you have a directory called `AspNetCoreTodo` inside another directory called `AspNetCoreTodo`. The top-level or "root" directory can contain one or more project directories. The root directory is sometimes called a **solution directory**. Later, you'll add more project directories side-by-side with the `AspNetCoreTodo` project directory, all within a single root solution directory.

You'll see quite a few files show up in the new project directory. Once you `cd` into the new directory, all you have to do is run the project:

```
dotnet run

Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Instead of printing to the console and exiting, this program starts a web server and waits for requests on port 5000.

Open your web browser and navigate to `http://localhost:5000`. You'll see the default ASP.NET Core splash page, which means your project is working! When you're done, press Ctrl-C in the terminal window to stop the server.

The parts of an ASP.NET Core project

The `dotnet new mvc` template generates a number of files and directories for you. Here are the most important things you get out of the box:

- The **Program.cs** and **Startup.cs** files set up the web server and ASP.NET Core pipeline. The `Startup` class is where you can add middleware that handles and modifies incoming requests, and serves things like static content or error pages. It's also where you add your own services to the dependency injection container (more on this later).

- The **Models**, **Views**, and **Controllers** directories contain the components of the Model-View-Controller (MVC) architecture. You'll explore all three in the next chapter.

- The **wwwroot** directory contains static assets like CSS, JavaScript, and image files. Files in `wwwroot` will be served as static content, and can be bundled and minified automatically.

- The **appsettings.json** file contains configuration settings ASP.NET Core will load on startup. You can use this to store database connection strings or other things that you don't want to hard-code.

Tips for Visual Studio Code

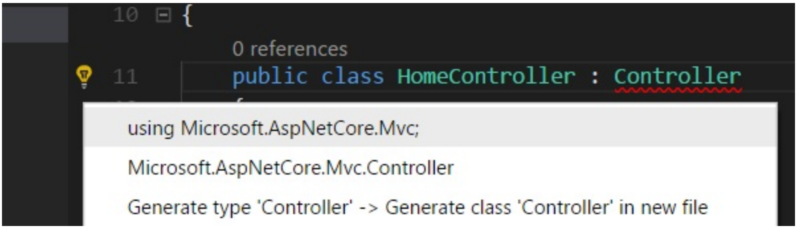
If you're using Visual Studio Code for the first time, here are a couple of helpful tips to get you started:

- **Open the project root folder**: In Visual Studio Code, choose File - Open or File - Open Folder. Open the `AspNetCoreTodo` folder (the root directory), not the inner project directory. If Visual Studio Code prompts you to install missing files, click Yes to add them.

- **F5 to run (and debug breakpoints)**: With your project open, press F5 to run the project in debug mode. This is the same as `dotnet run` on the command line, but you have the benefit of setting breakpoints in your code by clicking on the left margin:



- **Lightbulb to fix problems**: If your code contains red squiggles (compiler errors), put your cursor on the code that's red and look for the lightbulb icon on the left margin. The lightbulb menu will suggest common fixes, like adding a missing `using` statement to your code:



- **Compile quickly**: Use the shortcut `Command-Shift-B` or `Control-Shift-B` to run the Build task, which does the same thing as `dotnet build`.

> These tips apply to Visual Studio (not Code) on Windows too. If you're using Visual Studio, you'll need to open the `.csproj` project file directly. Visual Studio will later prompt you to save the Solution file, which you should save in the root directory (the first `AspNetCoreTodo` folder). You can also create an ASP.NET Core project directly within Visual Studio using the templates in File - New Project.

A note about Git

If you use Git or GitHub to manage your source code, now is a good time to do `git init` and initialize a Git repository in the project root directory:

```
cd ..
git init
```

Make sure you add a `.gitignore` file that ignores the `bin` and `obj` directories. The Visual Studio template on GitHub's gitignore template repo (https://github.com/github/gitignore) works great.

There's plenty more to explore, so let's dive in and start building an application!

MVC basics

In this chapter, you'll explore the MVC system in ASP.NET Core. **MVC** (Model-View-Controller) is a pattern for building web applications that's used in almost every web framework (Ruby on Rails and Express are popular examples), plus frontend JavaScript frameworks like Angular. Mobile apps on iOS and Android use a variation of MVC as well.

As the name suggests, MVC has three components: models, views, and controllers. **Controllers** handle incoming requests from a client or web browser and make decisions about what code to run. **Views** are templates (usually HTML plus a templating language like Handlebars, Pug, or Razor) that get data added to them and then are displayed to the user. **Models** hold the data that is added to views, or data that is entered by the user.

A common pattern for MVC code is:

- The controller receives a request and looks up some information in a database
- The controller creates a model with the information and attaches it to a view
- The view is rendered and displayed in the user's browser
- The user clicks a button or submits a form, which sends a

new request to the controller, and the cycle repeats

If you've worked with MVC in other languages, you'll feel right at home in ASP.NET Core MVC. If you're new to MVC, this chapter will teach you the basics and will help get you started.

What you'll build

The "Hello World" exercise of MVC is building a to-do list application. It's a great project since it's small and simple in scope, but it touches each part of MVC and covers many of the concepts you'd use in a larger application.

In this book, you'll build a to-do app that lets the user add items to their to-do list and check them off once complete. More specifically, you'll be creating:

- A web application server (sometimes called the "backend") using ASP.NET Core, C#, and the MVC pattern
- A database to store the user's to-do items using the SQLite database engine and a system called Entity Framework Core
- Web pages and an interface that the user will interact with via their browser, using HTML, CSS, and JavaScript (called the "frontend")
- A login form and security checks so each user's to-do list is kept private

Sound good? Let's built it! If you haven't already created a new ASP.NET Core project using `dotnet new mvc`, follow the steps in the previous chapter. You should be able to build and run the project and see the default welcome screen.

Create a controller

There are already a few controllers in the project's Controllers directory, including the `HomeController` that renders the default welcome screen you see when you visit `http://localhost:5000`. You can ignore these controllers for now.

Create a new controller for the to-do list functionality, called `TodoController`, and add the following code:

Controllers/TodoController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;

namespace AspNetCoreTodo.Controllers
{
    public class TodoController : Controller
    {
        // Actions go here
    }
}
```

Routes that are handled by controllers are called **actions**, and are represented by methods in the controller class. For example, the `HomeController` includes three action methods (`Index`, `About`, and `Contact`) which are mapped by ASP.NET Core to these route URLs:

```
localhost:5000/Home          -> Index()
localhost:5000/Home/About    -> About()
localhost:5000/Home/Contact -> Contact()
```

There are a number of conventions (common patterns) used by ASP.NET Core, such as the pattern that `FooController` becomes `/Foo`, and the `Index` action name can be left out of the URL. You can customize this behavior if you'd like, but for now, we'll stick to the default conventions.

Add a new action called `Index` to the `TodoController`, replacing the `// Actions go here` comment:

```
public class TodoController : Controller
{
    public IActionResult Index()
    {
        // Get to-do items from database

        // Put items into a model

        // Render view using the model
    }
}
```

Action methods can return views, JSON data, or HTTP status codes like `200 OK` and `404 Not Found`. The `IActionResult` return type gives you the flexibility to return any of these from the action.

It's a best practice to keep controllers as lightweight as possible. In this case, the controller will be responsible for getting the to-do items from the database, putting those items into a model the view can understand, and sending the view back to the user's browser.

Before you can write the rest of the controller code, you need to create a model and a view.

Create models

There are two separate model classes that need to be created: a model that represents a to-do item stored in the database (sometimes called an **entity**), and the model that will be combined with a view (the **MV** in MVC) and sent back to the user's browser. Because both of them can be referred to as "models", I'll refer to the latter as a **view model**.

First, create a class called `TodoItem` in the Models directory:

Models/TodoItem.cs

```
using System;
using System.ComponentModel.DataAnnotations;

namespace AspNetCoreTodo.Models
{
    public class TodoItem
    {
        public Guid Id { get; set; }

        public bool IsDone { get; set; }

        [Required]
        public string Title { get; set; }

        public DateTimeOffset? DueAt { get; set; }
    }
}
```

This class defines what the database will need to store for each to-do item: an ID, a title or name, whether the item is complete, and what the due date is. Each line defines a property of the class:

- The **Id** property is a guid, or a **g**lobally **u**nique **id**entifier. Guids (or GUIDs) are long strings of letters and numbers, like `43ec09f2-7f70-4f4b-9559-65011d5781bb`. Because guids are random and are extremely unlikely to be accidentally duplicated, they are commonly used as unique IDs. You could also use a number (integer) as a database entity ID, but you'd need to configure your database to

always increment the number when new rows are added to the database. Guids are generated randomly, so you don't have to worry about auto-incrementing.

- The **IsDone** property is a boolean (true/false value). By default, it will be `false` for all new items. Later you'll use write code to switch this property to `true` when the user clicks an item's checkbox in the view.
- The **Title** property is a string (text value). This will hold the name or description of the to-do item. The `[Required]` attribute tells ASP.NET Core that this string can't be null or empty.
- The **DueAt** property is a `DateTimeOffset`, which is a C# type that stores a date/time stamp along with a timezone offset from UTC. Storing the date, time, and timezone offset together makes it easy to render dates accurately on systems in different timezones.

Notice the `?` question mark after the `DateTimeOffset` type? That marks the DueAt property as **nullable**, or optional. If the `?` wasn't included, every to-do item would need to have a due date. The `Id` and `IsDone` properties aren't marked as nullable, so they are required and will always have a value (or a default value).

> Strings in C# are always nullable, so there's no need to mark the Title property as nullable. C# strings can be null, empty, or contain text.

Each property is followed by `get; set;`, which is a shorthand way of saying the property is read/write (or, more technically, it has a getter and setter methods).

At this point, it doesn't matter what the underlying database technology is. It could be SQL Server, MySQL, MongoDB, Redis, or something more exotic. This model defines what the database row or entry will look like in C# so you don't have to worry about the low-level database stuff in your code. This simple style of model is sometimes called a "plain old C# object" or POCO.

The view model

Often, the model (entity) you store in the database is similar but not *exactly* the same as the model you want to use in MVC (the view model). In this case, the `TodoItem` model represents a single item in the database, but the view might need to display two, ten, or a hundred to-do items (depending on how badly the user is procrastinating).

Because of this, the view model should be a separate class that holds an array of `TodoItem`s:

Models/TodoViewModel.cs

```
namespace AspNetCoreTodo.Models
{
    public class TodoViewModel
    {
        public TodoItem[] Items { get; set; }
    }
}
```

Now that you have some models, it's time to create a view that will take a `TodoViewModel` and render the right HTML to show the user their to-do list.

Create a view

Views in ASP.NET Core are built using the Razor templating language, which combines HTML and C# code. (If you've written pages using Handlebars moustaches, ERB in Ruby on Rails, or Thymeleaf in Java, you've already got the basic idea.)

Most view code is just HTML, with the occasional C# statement added in to pull data out of the view model and turn it into text or HTML. The C# statements are prefixed with the `@` symbol.

The view rendered by the `Index` action of the `TodoController` needs to take the data in the view model (a sequence of to-do items) and display it in a nice table for the user. By convention, views are placed in the `Views` directory, in a subdirectory corresponding to the controller name. The file name of the view is the name of the action with a `.cshtml` extension.

Create a `Todo` directory inside the `Views` directory, and add this file:

Views/Todo/Index.cshtml

```
@model TodoViewModel
```

```
@{
    ViewData["Title"] = "Manage your todo list";
}

<div class="panel panel-default todo-panel">
  <div class="panel-heading">@ViewData["Title"]</div>


  <table class="table table-hover">
      <thead>
          <tr>
              <td>&#x2714;</td>
              <td>Item</td>
              <td>Due</td>
          </tr>
      </thead>

      @foreach (var item in Model.Items)
      {
          <tr>
              <td>
                <input type="checkbox" class="done-c
heckbox">
              </td>
              <td>@item.Title</td>
              <td>@item.DueAt</td>
          </tr>
      }
  </table>

  <div class="panel-footer add-item-form">
    <!-- TODO: Add item form -->
  </div>
</div>
```

At the very top of the file, the `@model` directive tells Razor which model to expect this view to be bound to. The model is accessed through the `Model` property.

Assuming there are any to-do items in `Model.Items`, the `foreach` statement will loop over each to-do item and render a table row (`<tr>` element) containing the item's name and due date. A checkbox is also rendered that will let the user mark the item as complete.

The layout file

You might be wondering where the rest of the HTML is: what about the `<body>` tag, or the header and footer of the page? ASP.NET Core uses a layout view that defines the base structure that every other view is rendered inside of. It's stored in `Views/Shared/_Layout.cshtml`.

The default ASP.NET Core template includes Bootstrap and jQuery in this layout file, so you can quickly create a web application. Of course, you can use your own CSS and JavaScript libraries if you'd like.

Customizing the stylesheet

The default template also includes a stylesheet with some basic CSS rules. The stylesheet is stored in the `wwwroot/css` directory. Add a few new CSS style rules to the bottom of the `site.css` file:

wwwroot/css/site.css

```
div.todo-panel {
  margin-top: 15px;
}

table tr.done {
  text-decoration: line-through;
  color: #888;
}
```

You can use CSS rules like these to completely customize how your pages look and feel.

ASP.NET Core and Razor can do much more, such as partial views and server-rendered view components, but a simple layout and view is all you need for now. The official ASP.NET Core documentation (at https://docs.asp.net) contains a number of examples if you'd like to learn more.

Add a service class

You've created a model, a view, and a controller. Before you use the model and view in the controller, you also need to write code that will get the user's to-do items from a database.

You could write this database code directly in the controller, but it's a better practice to keep your code separate. Why? In a big, real-world application, you'll have to juggle many concerns:

- **Rendering views** and handling incoming data: this is what your controller already does.
- **Performing business logic**, or code and logic that's related to the purpose and "business" of your application. In a to-do list application, business logic means decisions like setting a default due date on new tasks, or only displaying tasks that are incomplete. Other examples of business logic include calculating a total cost based on product prices and tax rates, or checking whether a player has enough points to level up in a game.
- **Saving and retrieving** items from a database.

Again, it's possible to do all of these things in a single, massive controller, but that quickly becomes too hard to manage and test. Instead, it's common to see applications split up into two, three, or more "layers" or tiers that each handle one (and only

one) concern. This helps keep the controllers as simple as possible, and makes it easier to test and change the business logic and database code later.

Separating your application this way is sometimes called a **multi-tier** or **n-tier architecture**. In some cases, the tiers (layers) are isolated in completely separate projects, but other times it just refers to how the classes are organized and used. The important thing is thinking about how to split your application into manageable pieces, and avoid having controllers or bloated classes that try to do everything.

For this project, you'll use two application layers: a **presentation layer** made up of the controllers and views that interact with the user, and a **service layer** that contains business logic and database code. The presentation layer already exists, so the next step is to build a service that handles to-do business logic and saves to-do items to a database.

> Most larger projects use a 3-tier architecture: a presentation layer, a service logic layer, and a data repository layer. A **repository** is a class that's only focused on database code (no business logic). In this application, you'll combine these into a single service layer for simplicity, but feel free to experiment with different ways of architecting the code.

Create an interface

The C# language includes the concept of **interfaces**, where the definition of an object's methods and properties is separate from the class that actually contains the code for those methods and properties. Interfaces make it easy to keep your classes decoupled and easy to test, as you'll see here (and later in the *Automated testing* chapter). You'll use an interface to represent the service that can interact with to-do items in the database.

By convention, interfaces are prefixed with "I". Create a new file in the Services directory:

Services/ITodoItemService.cs

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using AspNetCoreTodo.Models;

namespace AspNetCoreTodo.Services
{
    public interface ITodoItemService
    {
        Task<TodoItem[]> GetIncompleteItemsAsync();
    }
}
```

Note that the namespace of this file is `AspNetCoreTodo.Services`. Namespaces are a way to organize .NET code files, and it's customary for the namespace to follow

the directory the file is stored in (`AspNetCoreTodo.Services` for files in the `Services` directory, and so on).

Because this file (in the `AspNetCoreTodo.Services` namespace) references the `TodoItem` class (in the `AspNetCoreTodo.Models` namespace), it needs to include a `using` statement at the top of the file to import that namespace. Without the `using` statement, you'll see an error like:

```
The type or namespace name 'TodoItem' could not be f
ound (are you missing a using directive or an assemb
ly reference?)
```

Since this is an interface, there isn't any actual code here, just the definition (or **method signature**) of the `GetIncompleteItemsAsync` method. This method requires no parameters and returns a `Task<TodoItem[]>` .

> If this syntax looks confusing, think: "a Task that contains an array of TodoItems".

The `Task` type is similar to a future or a promise, and it's used here because this method will be **asynchronous**. In other words, the method may not be able to return the list of to-do items right away because it needs to go talk to the database first. (More on this later.)

Create the service class

Now that the interface is defined, you're ready to create the actual service class. I'll cover database code in depth in the *Use a database* chapter, so for now you'll just fake it and always return two hard-coded items:

Services/FakeTodoItemService.cs

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using AspNetCoreTodo.Models;

namespace AspNetCoreTodo.Services
{
    public class FakeTodoItemService : ITodoItemService
    {
        public Task<TodoItem[]> GetIncompleteItemsAsync()
        {
            var item1 = new TodoItem
            {
                Title = "Learn ASP.NET Core",
                DueAt = DateTimeOffset.Now.AddDays(1)
            };

            var item2 = new TodoItem
            {
                Title = "Build awesome apps",
                DueAt = DateTimeOffset.Now.AddDays(2)
            };

            return Task.FromResult(new[] { item1, item2 });
        }
    }
}
```

This `FakeTodoItemService` implements the `ITodoItemService` interface but always returns the same array of two `TodoItem` s. You'll use this to test the controller and view, and then add real database code in *Use a database*.

Use dependency injection

Back in the `TodoController`, add some code to work with the `ITodoItemService`:

```
public class TodoController : Controller
{
    private readonly ITodoItemService _todoItemServi
ce;

    public TodoController(ITodoItemService todoItemS
ervice)
    {
        _todoItemService = todoItemService;
    }

    public IActionResult Index()
    {
        // Get to-do items from database

        // Put items into a model

        // Pass the view to a model and render
    }
}
```

Since `ITodoItemService` is in the `Services` namespace, you'll also need to add a `using` statement at the top:

```
using AspNetCoreTodo.Services;
```

The first line of the class declares a private variable to hold a reference to the `ITodoItemService`. This variable lets you use the service from the `Index` action method later (you'll see how in a minute).

The `public TodoController(ITodoItemService todoItemService)` line defines a **constructor** for the class. The constructor is a special method that is called when you want to create a new instance of a class (the `TodoController` class, in this case). By adding an `ITodoItemService` parameter to the constructor, you've declared that in order to create the `TodoController`, you'll need to provide an object that matches the `ITodoItemService` interface.

> Interfaces are awesome because they help decouple (separate) the logic of your application. Since the controller depends on the `ITodoItemService` interface, and not on any *specific* class, it doesn't know or care which class it's actually given. It could be the `FakeTodoItemService`, a different one that talks to a live database, or something else! As long as it matches the interface, the controller can use it. This makes it really easy to test parts of your application separately. I'll cover testing in detail in the *Automated testing* chapter.

Now you can finally use the `ITodoItemService` (via the private variable you declared) in your action method to get to-do items from the service layer:

```
public IActionResult Index()
{
    var items = await _todoItemService.GetIncomplete
ItemsAsync();

    // ...
}
```

Remember that the `GetIncompleteItemsAsync` method returned a `Task<TodoItem[]>`? Returning a `Task` means that the method won't necessarily have a result right away, but you can use the `await` keyword to make sure your code waits until the result is ready before continuing on.

The `Task` pattern is common when your code calls out to a database or an API service, because it won't be able to return a real result until the database (or network) responds. If you've used promises or callbacks in JavaScript or other languages, `Task` is the same idea: the promise that there will be a result - sometime in the future.

> If you've had to deal with "callback hell" in older JavaScript code, you're in luck. Dealing with asynchronous code in .NET is much easier thanks to the magic of the `await` keyword! `await` lets your code pause on an async operation, and then pick up where it left off when the underlying database or network request finishes. In the meantime, your application isn't blocked, because it can process other requests as needed. This pattern is simple but takes a little getting used to, so don't worry if this doesn't make sense right away. Just keep following along!

The only catch is that you need to update the `Index` method signature to return a `Task<IActionResult>` instead of just `IActionResult`, and mark it as `async`:

```
public async Task<IActionResult> Index()
{
    var items = await _todoItemService.GetIncomplete
ItemsAsync();

    // Put items into a model

    // Pass the view to a model and render
}
```

You're almost there! You've made the `TodoController` depend on the `ITodoItemService` interface, but you haven't yet told ASP.NET Core that you want the `FakeTodoItemService` to be

the actual service that's used under the hood. It might seem obvious right now since you only have one class that implements `ITodoItemService`, but later you'll have multiple classes that implement the same interface, so being explicit is necessary.

Declaring (or "wiring up") which concrete class to use for each interface is done in the `ConfigureServices` method of the `Startup` class. Right now, it looks something like this:

Startup.cs

```
public void ConfigureServices(IServiceCollection ser
vices)
{
    // (... some code)

    services.AddMvc();
}
```

The job of the `ConfigureServices` method is adding things to the **service container**, or the collection of services that ASP.NET Core knows about. The `services.AddMvc` line adds the services that the internal ASP.NET Core systems need (as an experiment, try commenting out this line). Any other services you want to use in your application must be added to the service container here in `ConfigureServices`.

Add the following line anywhere inside the `ConfigureServices` method:

```
services.AddSingleton<ITodoItemService, FakeTodoItem
Service>();
```

This line tells ASP.NET Core to use the `FakeTodoItemService` whenever the `ITodoItemService` interface is requested in a constructor (or anywhere else).

`AddSingleton` adds your service to the service container as a **singleton**. This means that only one copy of the `FakeTodoItemService` is created, and it's reused whenever the service is requested. Later, when you write a different service class that talks to a database, you'll use a different approach (called **scoped**) instead. I'll explain why in the *Use a database* chapter.

That's it! When a request comes in and is routed to the `TodoController`, ASP.NET Core will look at the available services and automatically supply the `FakeTodoItemService` when the controller asks for an `ITodoItemService`. Because the services are "injected" from the service container, this pattern is called **dependency injection**.

Finish the controller

The last step is to finish the controller code. The controller now has a list of to-do items from the service layer, and it needs to put those items into a `TodoViewModel` and bind that model to the view you created earlier:

Controllers/TodoController.cs

```
public async Task<IActionResult> Index()
{
    var items = await _todoItemService.GetIncomplete
ItemsAsync();

    var model = new TodoViewModel()
    {
        Items = items
    };

    return View(model);
}
```

If you haven't already, make sure these `using` statements are at the top of the file:

```
using AspNetCoreTodo.Services;
using AspNetCoreTodo.Models;
```

If you're using Visual Studio or Visual Studio Code, the editor will suggest these `using` statements when you put your cursor on a red squiggly line.

Test it out

To start the application, press F5 (if you're using Visual Studio or Visual Studio Code), or just type `dotnet run` in the terminal. If the code compiles without errors, the server will start up on port 5000 by default.

If your web browser didn't open automatically, open it and navigate to http://localhost:5000/todo. You'll see the view you created, with the data pulled from your fake database (for now).

Although it's possible to go directly to `http://localhost:5000/todo`, it would be nicer to add an item called **My to-dos** to the navbar. To do this, you can edit the shared layout file.

Update the layout

The layout file at `Views/Shared/_Layout.cshtml` contains the "base" HTML for each view. This includes the navbar, which is rendered at the top of each page.

To add a new item to the navbar, find the HTML code for the existing navbar items:

Views/Shared/_Layout.cshtml

```
<ul class="nav navbar-nav">
    <li><a asp-area="" asp-controller="Home" asp-act
ion="Index">
        Home
    </a></li>
    <li><a asp-area="" asp-controller="Home" asp-act
ion="About">
        About
    </a></li>
    <li><a asp-area="" asp-controller="Home" asp-act
ion="Contact">
        Contact
    </a></li>
</ul>
```

Add your own item that points to the `Todo` controller instead of `Home` :

```
<li>
    <a asp-controller="Todo" asp-action="Index">My t
o-dos</a>
</li>
```

The `asp-controller` and `asp-action` attributes on the `<a>` element are called **tag helpers**. Before the view is rendered, ASP.NET Core replaces these tag helpers with real HTML attributes. In this case, a URL to the `/Todo/Index` route is generated and added to the `<a>` element as an `href` attribute. This means you don't have to hard-code the route to the `TodoController`. Instead, ASP.NET Core generates it for you automatically.

> If you've used Razor in ASP.NET 4.x, you'll notice some syntax changes. Instead of using `@Html.ActionLink()` to generate a link to an action, tag helpers are now the recommended way to create links in your views. Tag helpers are useful for forms, too (you'll see why in a later chapter). You can learn about other tag helpers in the documentation at https://docs.asp.net.

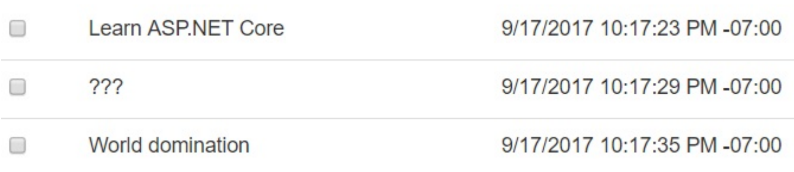
Add external packages

One of the big advantages of using a mature ecosystem like .NET is that the number of third-party packages and plugins is huge. Just like other package systems, you can download and install .NET packages that help with almost any task or problem you can imagine.

NuGet is both the package manager tool and the official package repository (at https://www.nuget.org). You can search for NuGet packages on the web, and install them from your local machine through the terminal (or the GUI, if you're using Visual Studio).

Install the Humanizer package

At the end of the last chapter, the to-do application displayed to-do items like this:



☐	Learn ASP.NET Core	9/17/2017 10:17:23 PM -07:00
☐	???	9/17/2017 10:17:29 PM -07:00
☐	World domination	9/17/2017 10:17:35 PM -07:00

The due date column is displaying dates in a format that's good for machines (called ISO 8601), but clunky for humans. Wouldn't it be nicer if it simply read "X days from now"?

You could write code yourself that converted an ISO 8601 date into a human-friendly string, but fortunately, there's a faster way.

The Humanizer package on NuGet solves this problem by providing methods that can "humanize" or rewrite almost anything: dates, times, durations, numbers, and so on. It's a fantastic and useful open-source project that's published under the permissive MIT license.

To add it to your project, run this command in the terminal:

```
dotnet add package Humanizer
```

If you peek at the `AspNetCoreTodo.csproj` project file, you'll see a new `PackageReference` line that references `Humanizer`.

Use Humanizer in the view

To use a package in your code, you usually need to add a `using` statement that imports the package at the top of the file.

Since Humanizer will be used to rewrite dates rendered in the view, you can use it directly in the view itself. First, add a `@using` statement at the top of the view:

Views/Todo/Index.cshtml

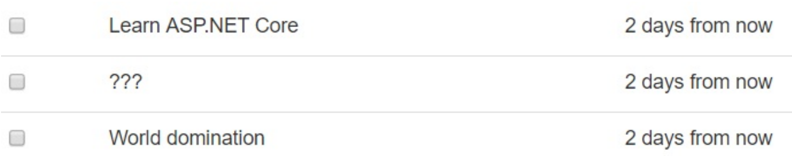
```
@model TodoViewModel
@using Humanizer

// ...
```

Then, update the line that writes the `DueAt` property to use Humanizer's `Humanize` method:

```
<td>@item.DueAt.Humanize()</td>
```

Now the dates are much more readable:



There are packages available on NuGet for everything from parsing XML to machine learning to posting to Twitter. ASP.NET Core itself, under the hood, is nothing more than a collection of NuGet packages that are added to your project.

> The project file created by `dotnet new mvc` includes a single reference to the `Microsoft.AspNetCore.All` package, which is a convenient "metapackage" that references all of the other ASP.NET Core packages you need for a typical project. That way, you don't need to have hundreds of package references in your project file.

In the next chapter, you'll use another set of NuGet packages (a system called Entity Framework Core) to write code that interacts with a database.

Use a database

Writing database code can be tricky. Unless you really know what you're doing, it's a bad idea to paste raw SQL query strings into your application code. An **object-relational mapper** (ORM) makes it easier to write code that interacts with a database by adding a layer of abstraction between your code and the database itself. Hibernate in Java and ActiveRecord in Ruby are two well-known ORMs.

There are a number of ORMs for .NET, including one built by Microsoft and included in ASP.NET Core by default: Entity Framework Core. Entity Framework Core makes it easy to connect to a number of different database types, and lets you use C# code to create database queries that are mapped back into C# models (POCOs).

> Remember how creating a service interface decoupled the controller code from the actual service class? Entity Framework Core is like a big interface over your database. Your C# code can stay database-agnostic, and you can swap out different providers depending on the underlying database technology.

Entity Framework Core can connect to relational databases like SQL Server, PostgreSQL, and MySQL, and also works with NoSQL (document) databases like Mongo. During development, you'll use SQLite in this project to make things easy to set up.

Connect to a database

There are a few things you need to use Entity Framework Core to connect to a database. Since you used `dotnet new` and the MVC + Individual Auth template to set your project, you've already got them:

- **The Entity Framework Core packages**. These are included by default in all ASP.NET Core projects.
- **A database** (naturally). The `app.db` file in the project root directory is a small SQLite database created for you by `dotnet new`. SQLite is a lightweight database engine that can run without requiring you to install any extra tools on your machine, so it's easy and quick to use in development.
- **A database context class**. The database context is a C# class that provides an entry point into the database. It's how your code will interact with the database to read and save items. A basic context class already exists in the `Data/ApplicationDbContext.cs` file.
- **A connection string**. Whether you are connecting to a local file database (like SQLite) or a database hosted elsewhere, you'll define a string that contains the name or

address of the database to connect to. This is already set up for you in the `appsettings.json` file: the connection string for the SQLite database is `DataSource=app.db`.

Entity Framework Core uses the database context, together with the connection string, to establish a connection to the database. You need to tell Entity Framework Core which context, connection string, and database provider to use in the `ConfigureServices` method of the `Startup` class. Here's what's defined for you, thanks to the template:

```
services.AddDbContext<ApplicationDbContext>(options
=>
    options.UseSqlite(
        Configuration.GetConnectionString("DefaultCo
nnection")));
```

This code adds the `ApplicationDbContext` to the service container, and tells Entity Framework Core to use the SQLite database provider, with the connection string from configuration (`appsettings.json`).

As you can see, `dotnet new` creates a lot of stuff for you! The database is set up and ready to be used. However, it doesn't have any tables for storing to-do items. In order to store your `TodoItem` entities, you'll need to update the context and migrate the database.

Update the context

There's not a whole lot going on in the database context yet:

Data/ApplicationDbContext.cs

```
public class ApplicationDbContext
             : IdentityDbContext<ApplicationUser>
{
    public ApplicationDbContext(
        DbContextOptions<ApplicationDbContext> optio
ns)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBui
lder builder)
    {
        base.OnModelCreating(builder);
        // ...
    }
}
```

Add a `DbSet` property to the `ApplicationDbContext`, right below the constructor:

```
public ApplicationDbContext(
    DbContextOptions<ApplicationDbContext> options)
    : base(options)
{
}

public DbSet<TodoItem> Items { get; set; }

// ...
```

A `DbSet` represents a table or collection in the database. By creating a `DbSet<TodoItem>` property called `Items`, you're telling Entity Framework Core that you want to store `TodoItem` entities in a table called `Items`.

You've updated the context class, but now there's one small problem: the context and database are now out of sync, because there isn't actually an `Items` table in the database. (Just updating the code of the context class doesn't change the database itself.)

In order to update the database to reflect the change you just made to the context, you need to create a **migration**.

> If you already have an existing database, search the web for "scaffold-dbcontext existing database" and read Microsoft's documentation on using the `Scaffold-DbContext` tool to reverse-engineer your database structure into the proper `DbContext` and model classes automatically.

Create a migration

Migrations keep track of changes to the database structure over time. They make it possible to undo (roll back) a set of changes, or create a second database with the same structure as the first. With migrations, you have a full history of modifications like adding or removing columns (and entire tables).

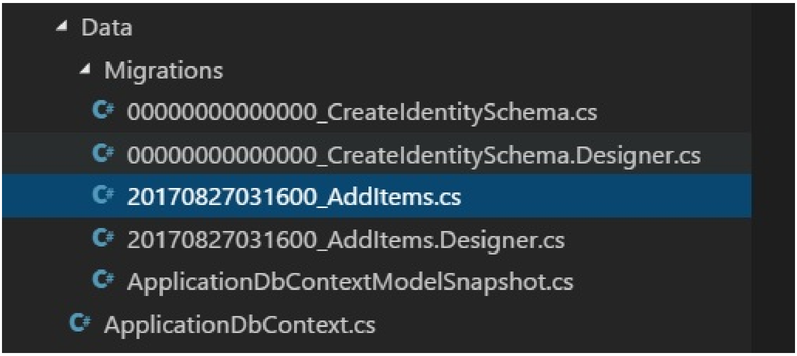
In the previous chapter, you added an `Items` set to the context. Since the context now includes a set (or table) that doesn't exist in the database, you need to create a migration to update the database:

```
dotnet ef migrations add AddItems
```

This creates a new migration called `AddItems` by examining any changes you've made to the context.

> If you get an error like `No executable found matching command "dotnet-ef"`, make sure you're in the right directory. These commands must be run from the project root directory (where the `Program.cs` file is).

If you open up the `Data/Migrations` directory, you'll see a few files:



The first migration file (with a name like `00_CreateIdentitySchema.cs`) was created and applied for you way back when you ran `dotnet new`. Your new `AddItem` migration is prefixed with a timestamp when you create it.

> You can see a list of migrations with `dotnet ef migrations list`.

If you open your migration file, you'll see two methods called `Up` and `Down`:

Data/Migrations/_AddItems.cs

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    // (... some code)

    migrationBuilder.CreateTable(
        name: "Items",
        columns: table => new
        {
```

```
                Id = table.Column<Guid>(nullable: false)
,
                DueAt = table.Column<DateTimeOffset>(nul
lable: true),
                IsDone = table.Column<bool>(nullable: fa
lse),
                Title = table.Column<string>(nullable: t
rue)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Items", x => x.Id);
            });

    // (some code...)
}

protected override void Down(MigrationBuilder migrat
ionBuilder)
{
    // (... some code)

    migrationBuilder.DropTable(
        name: "Items");

    // (some code...)
}
```

The `Up` method runs when you apply the migration to the database. Since you added a `DbSet<TodoItem>` to the database context, Entity Framework Core will create an `Items` table (with columns that match a `TodoItem`) when you apply the migration.

The `Down` method does the opposite: if you need to undo (roll back) the migration, the `Items` table will be dropped.

Workaround for SQLite limitations

There are some limitations of SQLite that get in the way if you try to run the migration as-is. Until this problem is fixed, use this workaround:

- Comment out or remove the `migrationBuilder.AddForeignKey` lines in the `Up` method.
- Comment out or remove any `migrationBuilder.DropForeignKey` lines in the `Down` method.

If you use a full-fledged SQL database, like SQL Server or MySQL, this won't be an issue and you won't need to do this (admittedly hackish) workaround.

Apply the migration

The final step after creating one (or more) migrations is to actually apply them to the database:

```
dotnet ef database update
```

This command will cause Entity Framework Core to create the `Items` table in the database.

> If you want to roll back the database, you can provide the name of the *previous* migration: `dotnet ef database update CreateIdentitySchema` This will run the `Down` methods of any migrations newer than the migration you specify.
>
> If you need to completely erase the database and start over, run `dotnet ef database drop` followed by `dotnet ef database update` to re-scaffold the database and bring it up to the current migration.

That's it! Both the database and the context are ready to go. Next, you'll use the context in your service layer.

Create a new service class

Back in the *MVC basics* chapter, you created a `FakeTodoItemService` that contained hard-coded to-do items. Now that you have a database context, you can create a new service class that will use Entity Framework Core to get the real items from the database.

Delete the `FakeTodoItemService.cs` file, and create a new file:

Services/TodoItemService.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using AspNetCoreTodo.Data;
using AspNetCoreTodo.Models;
using Microsoft.EntityFrameworkCore;

namespace AspNetCoreTodo.Services
{
    public class TodoItemService : ITodoItemService
    {
        private readonly ApplicationDbContext _context;

        public TodoItemService(ApplicationDbContext context)
        {
            _context = context;
        }

        public async Task<TodoItem[]> GetIncompleteItemsAsync()
        {
            var items = await _context.Items
                .Where(x => x.IsDone == false)
                .ToArrayAsync();
            return items;
        }
    }
}
```

You'll notice the same dependency injection pattern here that you saw in the *MVC basics* chapter, except this time it's the `ApplicationDbContext` that's getting injected. The `ApplicationDbContext` is already being added to the service container in the `ConfigureServices` method, so it's available for injection here.

Let's take a closer look at the code of the `GetIncompleteItemsAsync` method. First, it uses the `Items` property of the context to access all the to-do items in the `DbSet`:

```
var items = await _context.Items
```

Then, the `Where` method is used to filter only the items that are not complete:

```
.Where(x => x.IsDone == false)
```

The `Where` method is a feature of C# called LINQ (**l**anguage **in**tegrated **q**uery), which takes inspiration from functional programming and makes it easy to express database queries in code. Under the hood, Entity Framework Core translates the `Where` method into a statement like `SELECT * FROM Items WHERE IsDone = 0`, or an equivalent query document in a NoSQL database.

Finally, the `ToArrayAsync` method tells Entity Framework Core to get all the entities that matched the filter and return them as an array. The `ToArrayAsync` method is asynchronous (it returns a `Task`), so it must be `await` ed to get its value.

To make the method a little shorter, you can remove the intermediate `items` variable and just return the result of the query directly (which does the same thing):

```
public async Task<TodoItem[]> GetIncompleteItemsAsyn
c()
{
    return await _context.Items
        .Where(x => x.IsDone == false)
        .ToArrayAsync();
}
```

Update the service container

Because you deleted the `FakeTodoItemService` class, you'll need to update the line in `ConfigureServices` that is wiring up the `ITodoItemService` interface:

```
services.AddScoped<ITodoItemService, TodoItemService
>();
```

`AddScoped` adds your service to the service container using the **scoped** lifecycle. This means that a new instance of the `TodoItemService` class will be created during each web request. This is required for service classes that interact with a database.

> Adding a service class that interacts with Entity Framework Core (and your database) with the singleton lifecycle (or other lifecycles) can cause problems, because of how Entity Framework Core manages database connections per request under the hood. To avoid that, always use the scoped lifecycle for services that interact with Entity Framework Core.

The `TodoController` that depends on an injected `ITodoItemService` will be blissfully unaware of the change in services classes, but under the hood it'll be using Entity Framework Core and talking to a real database!

Test it out

Start up the application and navigate to `http://localhost:5000/todo`. The fake items are gone, and your application is making real queries to the database. There doesn't happen to be any saved to-do items, so it's blank for now.

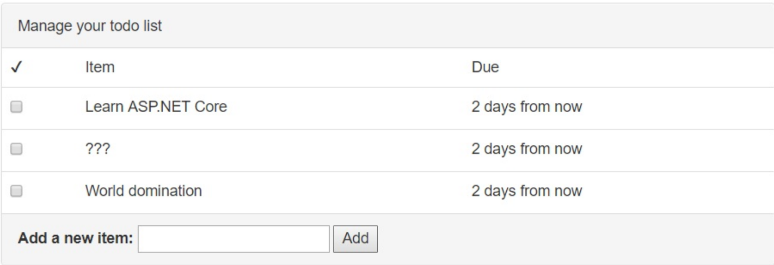
In the next chapter, you'll add more features to the application, starting with the ability to create new to-do items.

Add more features

Now that you've connected to a database using Entity Framework Core, you're ready to add some more features to the application. First, you'll make it possible to add new to-do items using a form.

Add new to-do items

The user will add new to-do items with a simple form below the list:



Adding this feature requires a few steps:

- Adding a form to the view
- Creating a new action on the controller to handle the form
- Adding code to the service layer to update the database

Add a form

The `Views/Todo/Index.cshtml` view has a placeholder for the Add Item form:

```
<div class="panel-footer add-item-form">
  <!-- TODO: Add item form -->
</div>
```

To keep things separate and organized, you'll create the form as a **partial view**. A partial view is a small piece of a larger view that lives in a separate file.

Create an `AddItemPartial.cshtml` view:

Views/Todo/AddItemPartial.cshtml

```
@model TodoItem

<form asp-action="AddItem" method="POST">
    <label asp-for="Title">Add a new item:</label>
    <input asp-for="Title">
    <button type="submit">Add</button>
</form>
```

The `asp-action` tag helper can generate a URL for the form, just like when you use it on an `<a>` element. In this case, the `asp-action` helper gets replaced with the real path to the `AddItem` route you'll create:

```
<form action="/Todo/AddItem" method="POST">
```

Adding an `asp-` tag helper to the `<form>` element also adds a hidden field to the form containing a verification token. This verification token can be used to prevent cross-site request forgery (CSRF) attacks. You'll verify the token when you write the action.

That takes care of creating the partial view. Now, reference it from the main Todo view:

Views/Todo/Index.cshtml

```
<div class="panel-footer add-item-form">
  @await Html.PartialAsync("AddItemPartial", new TodoItem())
</div>
```

Add an action

When a user clicks Add on the form you just created, their browser will construct a POST request to `/Todo/AddItem` on your application. That won't work right now, because there isn't any action that can handle the `/Todo/AddItem` route. If you try it now, ASP.NET Core will return a `404 Not Found` error.

You'll need to create a new action called `AddItem` on the `TodoController` :

```
[ValidateAntiForgeryToken]
public async Task<IActionResult> AddItem(TodoItem ne
wItem)
{
    if (!ModelState.IsValid)
    {
        return RedirectToAction("Index");
    }

    var successful = await _todoItemService.AddItemA
sync(newItem);
    if (!successful)
    {
        return BadRequest("Could not add item.");
    }

    return RedirectToAction("Index");
}
```

Notice how the new `AddItem` action accepts a `TodoItem` parameter? This is the same `TodoItem` model you created in the *MVC basics* chapter to store information about a to-do item. When it's used here as an action parameter, ASP.NET Core will automatically perform a process called **model binding**.

Model binding looks at the data in a request and tries to intelligently match the incoming fields with properties on the model. In other words, when the user submits this form and

their browser POSTs to this action, ASP.NET Core will grab the information from the form and place it in the `newItem` variable.

The `[ValidateAntiForgeryToken]` attribute before the action tells ASP.NET Core that it should look for (and verify) the hidden verification token that was added to the form by the `asp-action` tag helper. This is an important security measure to prevent cross-site request forgery (CSRF) attacks, where your users could be tricked into submitting data from a malicious site. The verification token ensures that your application is actually the one that rendered and submitted the form.

Take a look at the `AddItemPartial.cshtml` view once more. The `@model TodoItem` line at the top of the file tells ASP.NET Core that the view should expect to be paired with the `TodoItem` model. This makes it possible to use `asp-for="Title"` on the `<input>` tag to let ASP.NET Core know that this input element is for the `Title` property.

Because of the `@model` line, the partial view will expect to be passed a `TodoItem` object when it's rendered. Passing it a `new TodoItem` via `Html.PartialAsync` initializes the form with an empty item. (Try appending `{ Title = "hello" }` and see what happens!)

During model binding, any model properties that can't be matched up with fields in the request are ignored. Since the form only includes a `Title` input element, you can expect that the other properties on `TodoItem` (the `IsDone` flag, the `DueAt` date) will be empty or contain default values.

> Instead of reusing the `TodoItem` model, another approach would be to create a separate model (like `NewTodoItem`) that's only used for this action and only has the specific properties (Title) you need for adding a new to-do item. Model binding is still used, but this way you've separated the model that's used for storing a to-do item in the database from the model that's used for binding incoming request data. This is sometimes called a **binding model** or a **data transfer object** (DTO). This pattern is common in larger, more complex projects.

After binding the request data to the model, ASP.NET Core also performs **model validation**. Validation checks whether the data bound to the model from the incoming request makes sense or is valid. You can add attributes to the model to tell ASP.NET Core how it should be validated.

The `[Required]` attribute on the `Title` property tells ASP.NET Core's model validator to consider the title invalid if it is missing or blank. Take a look at the code of the `AddItem`

action: the first block checks whether the `ModelState` (the model validation result) is valid. It's customary to do this validation check right at the beginning of the action:

```
if (!ModelState.IsValid)
{
    return RedirectToAction("Index");
}
```

If the `ModelState` is invalid for any reason, the browser will be redirected to the `/Todo/Index` route, which refreshes the page.

Next, the controller calls into the service layer to do the actual database operation of saving the new to-do item:

```
var successful = await _todoItemService.AddItemAsync
(newItem);
if (!successful)
{
    return BadRequest(new { error = "Could not add i
tem." });
}
```

The `AddItemAsync` method will return `true` or `false` depending on whether the item was successfully added to the database. If it fails for some reason, the action will return an HTTP `400 Bad Request` error along with an object that contains an error message.

Finally, if everything completed without errors, the action redirects the browser to the `/Todo/Index` route, which refreshes the page and displays the new, updated list of to-do items to the user.

Add a service method

If you're using a code editor that understands C#, you'll see red squiggely lines under `AddItemAsync` because the method doesn't exist yet.

As a last step, you need to add a method to the service layer. First, add it to the interface definition in `ITodoItemService`:

```
public interface ITodoItemService
{
    Task<TodoItem[]> GetIncompleteItemsAsync();

    Task<bool> AddItemAsync(TodoItem newItem);
}
```

Then, the actual implementation in `TodoItemService`:

```
public async Task<bool> AddItemAsync(TodoItem newItem
)
{
    newItem.Id = Guid.NewGuid();
    newItem.IsDone = false;
    newItem.DueAt = DateTimeOffset.Now.AddDays(3);

    _context.Items.Add(newItem);

    var saveResult = await _context.SaveChangesAsync
();
    return saveResult == 1;
}
```

The `newItem.Title` property has already been set by ASP.NET Core's model binder, so this method only needs to assign an ID and set the default values for the other properties. Then, the new item is added to the database context. It isn't actually saved until you call `SaveChangesAsync()`. If the save operation was successful, `SaveChangesAsync()` will return 1.

Try it out

Run the application and add some items to your to-do list with the form. Since the items are being stored in the database, they'll still be there even after you stop and start the application again.

> As an extra challenge, try adding a date picker using HTML and JavaScript, and let the user choose an (optional) date for the `DueAt` property. Then, use that date instead of always making new tasks that are due in 3 days.

Complete items with a checkbox

Adding items to your to-do list is great, but eventually you'll need to get things done, too. In the `Views/Todo/Index.cshtml` view, a checkbox is rendered for each to-do item:

```
<input type="checkbox" class="done-checkbox">
```

Clicking the checkbox doesn't do anything (yet). Just like the last chapter, you'll add this behavior using forms and actions. In this case, you'll also need a tiny bit of JavaScript code.

Add form elements to the view

First, update the view and wrap each checkbox with a `<form>` element. Then, add a hidden element containing the item's ID:

Views/Todo/Index.cshtml

```
<td>
    <form asp-action="MarkDone" method="POST">
        <input type="checkbox" class="done-checkbox">

        <input type="hidden" name="id" value="@item.
Id">
    </form>
</td>
```

When the `foreach` loop runs in the view and prints a row for each to-do item, a copy of this form will exist in each row. The hidden input containing the to-do item's ID makes it possible for your controller code to tell which box was checked. (Without it, you'd be able to tell that *some* box was checked, but not which one.)

If you run your application right now, the checkboxes still won't do anything, because there's no submit button to tell the browser to create a POST request with the form's data. You could add a submit button under each checkbox, but that would be a silly user experience. Ideally, clicking the checkbox should automatically submit the form. You can achieve that by adding some JavaScript.

Add JavaScript code

Find the `site.js` file in the `wwwroot/js` directory and add this code:

wwwroot/js/site.js

```
$(document).ready(function() {

    // Wire up all of the checkboxes to run markComp
leted()
    $('.done-checkbox').on('click', function(e) {
        markCompleted(e.target);
    });
});

function markCompleted(checkbox) {
    checkbox.disabled = true;

    var row = checkbox.closest('tr');
    $(row).addClass('done');

    var form = checkbox.closest('form');
    form.submit();
}
```

This code first uses jQuery (a JavaScript helper library) to attach some code to the `click` even of all the checkboxes on the page with the CSS class `done-checkbox`. When a checkbox is clicked, the `markCompleted()` function is run.

The `markCompleted()` function does a few things:

- Adds the `disabled` attribute to the checkbox so it can't be

clicked again
- Adds the `done` CSS class to the parent row that contains the checkbox, which changes the way the row looks based on the CSS rules in `style.css`
- Submits the form

That takes care of the view and frontend code. Now it's time to add a new action!

Add an action to the controller

As you've probably guessed, you need to add an action called `MarkDone` in the `TodoController`:

```
[ValidateAntiForgeryToken]
public async Task<IActionResult> MarkDone(Guid id)
{
    if (id == Guid.Empty)
    {
        return RedirectToAction("Index");
    }

    var successful = await _todoItemService.MarkDone
Async(id);
    if (!successful)
    {
        return BadRequest("Could not mark item as do
ne.");
    }

    return RedirectToAction("Index");
}
```

Let's step through each line of this action method. First, the method accepts a `Guid` parameter called `id` in the method signature. Unlike the `AddItem` action, which used a model and model binding/validation, the `id` parameter is very simple. If the incoming request data includes a field called `id`, ASP.NET Core will try to parse it as a guid. This works because the hidden element you added to the checkbox form is named `id`.

Since you aren't using model binding, there's no `ModelState` to check for validity. Instead, you can check the guid value directly to make sure it's valid. If for some reason the `id`

parameter in the request was missing or couldn't be parsed as a guid, `id` will have a value of `Guid.Empty`. If that's the case, the action tells the browser to redirect to `/Todo/Index` and refresh the page.

Next, the controller needs to call the service layer to update the database. This will be handled by a new method called `MarkDoneAsync` on the `ITodoItemService` interface, which will return true or false depending on whether the update succeeded:

```
var successful = await _todoItemService.MarkDoneAsyn
c(id);
if (!successful)
{
    return BadRequest("Could not mark item as done."
);
}
```

Finally, if everything looks good, the browser is redirected to the `/Todo/Index` action and the page is refreshed.

With the view and controller updated, all that's left is adding the missing service method.

Add a service method

First, add `MarkDoneAsync` to the interface definition:

Services/ITodoItemService.cs

```
Task<bool> MarkDoneAsync(Guid id);
```

Then, add the concrete implementation to the `TodoItemService` :

Services/TodoItemService.cs

```
public async Task<bool> MarkDoneAsync(Guid id)
{
    var item = await _context.Items
        .Where(x => x.Id == id)
        .SingleOrDefaultAsync();

    if (item == null) return false;

    item.IsDone = true;

    var saveResult = await _context.SaveChangesAsync
();
    return saveResult == 1; // One entity should hav
e been updated
}
```

This method uses Entity Framework Core and `Where()` to find an item by ID in the database. The `SingleOrDefaultAsync()` method will either return the item or `null` if it couldn't be found.

Once you're sure that `item` isn't null, it's a simple matter of setting the `IsDone` property:

```
item.IsDone = true;
```

Changing the property only affects the local copy of the item until `SaveChangesAsync()` is called to persist the change back to the database. `SaveChangesAsync()` returns a number that indicates how many entities were updated during the save operation. In this case, it'll either be 1 (the item was updated) or 0 (something went wrong).

Try it out

Run the application and try checking some items off the list. Refresh the page and they'll disappear completely, because of the `Where()` filter in the `GetIncompleteItemsAsync()` method.

Right now, the application contains a single, shared to-do list. It'd be even more useful if it kept track of individual to-do lists for each user. In the next chapter, you'll add login and security features to the project.

Security and identity

Security is a major concern of any modern web application or API. It's important to keep your user or customer data safe and out of the hands of attackers. This is a very broad topic, involving things like:

- Sanitizing data input to prevent SQL injection attacks
- Preventing cross-domain (CSRF) attacks in forms
- Using HTTPS (connection encryption) so data can't be intercepted as it travels over the Internet
- Giving users a way to securely sign in with a password or other credentials
- Designing password reset, account recovery, and multi-factor authentication flows

ASP.NET Core can help make all of this easier to implement. The first two (protection against SQL injection and cross-domain attacks) are already built-in, and you can add a few lines of code to enable HTTPS support. This chapter will mainly focus on the **identity** aspects of security: handling user accounts, authenticating (logging in) your users securely, and making authorization decisions once they are authenticated.

> Authentication and authorization are distinct ideas that are often confused. **Authentication** deals with whether a user is logged in, while **authorization** deals with what they are allowed to do *after* they log in. You can think of authentication as asking the question, "Do I know who this user is?" While authorization asks, "Does this user have permission to do *X*?"

The MVC + Individual Authentication template you used to scaffold the project includes a number of classes built on top of ASP.NET Core Identity, an authentication and identity system that's part of ASP.NET Core. Out of the box, this adds the ability to log in with an email and password.

What is ASP.NET Core Identity?

ASP.NET Core Identity is the identity system that ships with ASP.NET Core. Like everything else in the ASP.NET Core ecosystem, it's a set of NuGet packages that can be installed in any project (and are already included if you use the default template).

ASP.NET Core Identity takes care of storing user accounts, hashing and storing passwords, and managing roles for users. It supports email/password login, multi-factor authentication,

social login with providers like Google and Facebook, as well as connecting to other services using protocols like OAuth 2.0 and OpenID Connect.

The Register and Login views that ship with the MVC + Individual Authentication template already take advantage of ASP.NET Core Identity, and they already work! Try registering for an account and logging in.

Require authentication

Often you'll want to require the user to log in before they can access certain parts of your application. For example, it makes sense to show the home page to everyone (whether you're logged in or not), but only show your to-do list after you've logged in.

You can use the `[Authorize]` attribute in ASP.NET Core to require a logged-in user for a particular action, or an entire controller. To require authentication for all actions of the `TodoController`, add the attribute above the first line of the controller:

Controllers/TodoController.cs

```
[Authorize]
public class TodoController : Controller
{
    // ...
}
```

Add this `using` statement at the top of the file:

```
using Microsoft.AspNetCore.Authorization;
```

Try running the application and accessing `/todo` without being logged in. You'll be redirected to the login page automatically.

> The `[Authorize]` attribute is actually doing an authentication check here, not an authorization check (despite the name of the attribute). Later, you'll use the attribute to check **both** authentication and authorization.

Using identity in the application

The to-do list items themselves are still shared between all users, because the stored to-do entities aren't tied to a particular user. Now that the `[Authorize]` attribute ensures that you must be logged in to see the to-do view, you can filter the database query based on who is logged in.

First, inject a `UserManager<ApplicationUser>` into the `TodoController`:

Controllers/TodoController.cs

```
[Authorize]
public class TodoController : Controller
{
    private readonly ITodoItemService _todoItemServi
ce;
    private readonly UserManager<ApplicationUser> _u
serManager;

    public TodoController(ITodoItemService todoItemS
ervice,
        UserManager<ApplicationUser> userManager)
    {
        _todoItemService = todoItemService;
        _userManager = userManager;
    }

    // ...
}
```

You'll need to add a new `using` statement at the top:

```
using Microsoft.AspNetCore.Identity;
```

The `UserManager` class is part of ASP.NET Core Identity. You can use it to get the current user in the `Index` action:

```
public async Task<IActionResult> Index()
{
    var currentUser = await _userManager.GetUserAsyn
c(User);
    if (currentUser == null) return Challenge();

    var items = await _todoItemService
        .GetIncompleteItemsAsync(currentUser);

    var model = new TodoViewModel()
    {
        Items = items
    };

    return View(model);
}
```

The new code at the top of the action method uses the `UserManager` to look up the current user from the `User` property available in the action:

```
var currentUser = await _userManager.GetUserAsync(Us
er);
```

If there is a logged-in user, the `User` property contains a lightweight object with some (but not all) of the user's information. The `UserManager` uses this to look up the full user details in the database via the `GetUserAsync()` method.

The value of `currentUser` should never be null, because the `[Authorize]` attribute is present on the controller. However, it's a good idea to do a sanity check, just in case. You can use the `Challenge()` method to force the user to log in again if their information is missing:

```
if (currentUser == null) return Challenge();
```

Since you're now passing an `ApplicationUser` parameter to `GetIncompleteItemsAsync()`, you'll need to update the `ITodoItemService` interface:

Services/ITodoItemService.cs

```
public interface ITodoItemService
{
    Task<TodoItem[]> GetIncompleteItemsAsync(
        ApplicationUser user);

    // ...
}
```

Since you changed the `ITodoItemService` interface, you also need to update the signature of the `GetIncompleteItemsAsync()` method in the `TodoItemService`:

Services/TodoItemService

```
public async Task<TodoItem[]> GetIncompleteItemsAsyn
c(
    ApplicationUser user)
```

The next step is to update the database query and add a filter to show only the items created by the current user. Before you can do that, you need to add a new property to the database.

Update the database

You'll need to add a new property to the `TodoItem` entity model so each item can "remember" the user that owns it:

Models/TodoItem.cs

```
public string UserId { get; set; }
```

Since you updated the entity model used by the database context, you also need to migrate the database. Create a new migration using `dotnet ef` in the terminal:

```
dotnet ef migrations add AddItemUserId
```

This creates a new migration called `AddItemUserId` which will add a new column to the `Items` table, mirroring the change you made to the `TodoItem` model.

Use `dotnet ef` again to apply it to the database:

```
dotnet ef database update
```

Update the service class

With the database and the database context updated, you can now update the `GetIncompleteItemsAsync()` method in the `TodoItemService` and add another clause to the `Where` statement:

Services/TodoItemService.cs

```
public async Task<TodoItem[]> GetIncompleteItemsAsyn
c(
    ApplicationUser user)
{
    return await _context.Items
        .Where(x => x.IsDone == false && x.UserId ==
 user.Id)
        .ToArrayAsync();
}
```

If you run the application and register or log in, you'll see an empty to-do list once again. Unfortunately, any items you try to add disappear into the ether, because you haven't updated the `AddItem` action to be user-aware yet.

Update the AddItem and MarkDone actions

You'll need to use the `UserManager` to get the current user in the `AddItem` and `MarkDone` action methods, just like you did in `Index`.

Here are both updated methods:

Controllers/TodoController.cs

```
[ValidateAntiForgeryToken]
public async Task<IActionResult> AddItem(TodoItem ne
wItem)
{
    if (!ModelState.IsValid)
    {
        return RedirectToAction("Index");
    }

    var currentUser = await _userManager.GetUserAsyn
c(User);
    if (currentUser == null) return Challenge();

    var successful = await _todoItemService
        .AddItemAsync(newItem, currentUser);

    if (!successful)
    {
        return BadRequest("Could not add item.");
    }

    return RedirectToAction("Index");
```

```
}

[ValidateAntiForgeryToken]
public async Task<IActionResult> MarkDone(Guid id)
{
    if (id == Guid.Empty)
    {
        return RedirectToAction("Index");
    }

    var currentUser = await _userManager.GetUserAsyn
c(User);
    if (currentUser == null) return Challenge();

    var successful = await _todoItemService
        .MarkDoneAsync(id, currentUser);

    if (!successful)
    {
        return BadRequest("Could not mark item as do
ne.");
    }

    return RedirectToAction("Index");
}
```

Both service methods must now accept an `ApplicationUser` parameter. Update the interface definition in `ITodoItemService`:

```
Task<bool> AddItemAsync(TodoItem newItem, Applicatio
nUser user);

Task<bool> MarkDoneAsync(Guid id, ApplicationUser us
er);
```

And finally, update the service method implementations in the `TodoItemService`. In `AddItemAsync` method, set the `UserId` property when you construct a `new TodoItem`:

```
public async Task<bool> AddItemAsync(
    TodoItem newItem, ApplicationUser user)
{
    newItem.Id = Guid.NewGuid();
    newItem.IsDone = false;
    newItem.DueAt = DateTimeOffset.Now.AddDays(3);
    newItem.UserId = user.Id;

    // ...
}
```

The `Where` clause in the `MarkDoneAsync` method also needs to check for the user's ID, so a rogue user can't complete someone else's items by guessing their IDs:

```
public async Task<bool> MarkDoneAsync(
    Guid id, ApplicationUser user)
{
    var item = await _context.Items
        .Where(x => x.Id == id && x.UserId == user.I
d)
        .SingleOrDefaultAsync();

    // ...
}
```

All done! Try using the application with two different user accounts. The to-do items stay private for each account.

Authorization with roles

Roles are a common approach to handling authorization and permissions in a web application. For example, it's common to create an Administrator role that gives admin users more permissions or power than normal users.

In this project, you'll add a Manage Users page that only administrators can see. If normal users try to access it, they'll see an error.

Add a Manage Users page

First, create a new controller:

Controllers/ManageUsersController.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using AspNetCoreTodo.Models;
using Microsoft.EntityFrameworkCore;

namespace AspNetCoreTodo.Controllers
{
    [Authorize(Roles = "Administrator")]
```

```
    public class ManageUsersController : Controller
    {
        private readonly UserManager<ApplicationUser
>
            _userManager;

        public ManageUsersController(
            UserManager<ApplicationUser> userManager
)
        {
            _userManager = userManager;
        }

        public async Task<IActionResult> Index()
        {
            var admins = (await _userManager
                .GetUsersInRoleAsync("Administrator"
))
                .ToArray();

            var everyone = await _userManager.Users
                .ToArrayAsync();

            var model = new ManageUsersViewModel
            {
                Administrators = admins,
                Everyone = everyone
            };

            return View(model);
        }
    }
}
```

Setting the `Roles` property on the `[Authorize]` attribute will ensure that the user must be logged in **and** assigned the Administrator role in order to view the page.

Next, create a view model:

Models/ManageUsersViewModel.cs

```
using System.Collections.Generic;

namespace AspNetCoreTodo.Models
{
    public class ManageUsersViewModel
    {
        public ApplicationUser[] Administrators { get
; set; }

        public ApplicationUser[] Everyone { get; set
;}
    }
}
```

Finally, create a `Views/ManageUsers` folder and a view for the `Index` action:

Views/ManageUsers/Index.cshtml

```
@model ManageUsersViewModel

@{
    ViewData["Title"] = "Manage users";
```

```
}

<h2>@ViewData["Title"]</h2>

<h3>Administrators</h3>

<table class="table">
    <thead>
        <tr>
            <td>Id</td>
            <td>Email</td>
        </tr>
    </thead>

    @foreach (var user in Model.Administrators)
    {
        <tr>
            <td>@user.Id</td>
            <td>@user.Email</td>
        </tr>
    }
</table>

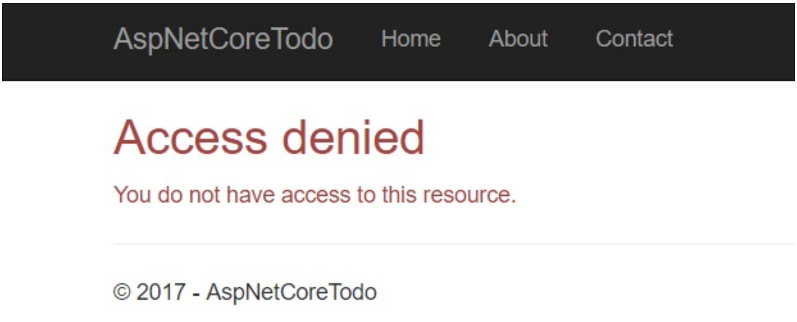
<h3>Everyone</h3>

<table class="table">
    <thead>
        <tr>
            <td>Id</td>
            <td>Email</td>
        </tr>
    </thead>

    @foreach (var user in Model.Everyone)
    {
```

```
            <tr>
                <td>@user.Id</td>
                <td>@user.Email</td>
            </tr>
        }
    </table>
```

Start up the application and try to access the `/ManageUsers` route while logged in as a normal user. You'll see this access denied page:



That's because users aren't assigned the Administrator role automatically.

Create a test administrator account

For obvious security reasons, it isn't possible for anyone to register a new administrator account themselves. In fact, the Administrator role doesn't even exist in the database yet!

You can add the Administrator role plus a test administrator account to the database the first time the application starts up. Adding first-time data to the database is called initializing or **seeding** the database.

Create a new class in the root of the project called `SeedData`:

SeedData.cs

```
using System;
using System.Linq;
using System.Threading.Tasks;
using AspNetCoreTodo.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;

namespace AspNetCoreTodo
{
    public static class SeedData
    {
        public static async Task InitializeAsync(
            IServiceProvider services)
        {
            var roleManager = services
                .GetRequiredService<RoleManager<Iden
tityRole>>();
            await EnsureRolesAsync(roleManager);

            var userManager = services
                .GetRequiredService<UserManager<Appl
icationUser>>();
            await EnsureTestAdminAsync(userManager);
        }
    }
}
```

The `InitializeAsync()` method uses an `IServiceProvider` (the collection of services that is set up in the `Startup.ConfigureServices()` method) to get the `RoleManager` and `UserManager` from ASP.NET Core Identity.

Add two more methods below the `InitializeAsync()` method. First, the `EnsureRolesAsync()` method:

```
private static async Task EnsureRolesAsync(
    RoleManager<IdentityRole> roleManager)
{
    var alreadyExists = await roleManager
        .RoleExistsAsync(Constants.AdministratorRole
);

    if (alreadyExists) return;

    await roleManager.CreateAsync(
        new IdentityRole(Constants.AdministratorRole
));
}
```

This method checks to see if an `Administrator` role exists in the database. If not, it creates one. Instead of repeatedly typing the string `"Administrator"`, create a small class called `Constants` to hold the value:

Constants.cs

```
namespace AspNetCoreTodo
{
    public static class Constants
    {
        public const string AdministratorRole = "Adm
inistrator";
    }
}
```

> If you want, you can update the `ManageUsersController` to use this constant value as well.

Next, write the `EnsureTestAdminAsync()` method:

SeedData.cs

```
private static async Task EnsureTestAdminAsync(
    UserManager<ApplicationUser> userManager)
{
    var testAdmin = await userManager.Users
        .Where(x => x.UserName == "admin@todo.local"
)
        .SingleOrDefaultAsync();

    if (testAdmin != null) return;

    testAdmin = new ApplicationUser
    {
        UserName = "admin@todo.local",
        Email = "admin@todo.local"
    };
    await userManager.CreateAsync(
        testAdmin, "NotSecure123!!");
    await userManager.AddToRoleAsync(
        testAdmin, Constants.AdministratorRole);
}
```

If there isn't already a user with the username `admin@todo.local` in the database, this method will create one and assign a temporary password. After you log in for the first time, you should change the account's password to something secure!

Next, you need to tell your application to run this logic when it starts up. Modify `Program.cs` and update `Main()` to call a new method, `InitializeDatabase()`:

Program.cs

```
public static void Main(string[] args)
{
    var host = BuildWebHost(args);
    InitializeDatabase(host);
    host.Run();
}
```

Then, add the new method to the class below `Main()`:

```
private static void InitializeDatabase(IWebHost host
)
{
    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;

        try
        {
            SeedData.InitializeAsync(services).Wait(
);
        }
        catch (Exception ex)
        {
            var logger = services
                .GetRequiredService<ILogger<Program>
>();
            logger.LogError(ex, "Error occurred seed
ing the DB.");
        }
    }
}
```

Add this `using` statement to the top of the file:

```
using Microsoft.Extensions.DependencyInjection;
```

This method gets the service collection that `SeedData.InitializeAsync()` needs and then runs the method to seed the database. If something goes wrong, an error is logged.

> Because `InitializeAsync()` returns a `Task`, the `Wait()` method must be used to make sure it finishes before the application starts up. You'd normally use `await` for this, but for technical reasons you can't use `await` in the `Program` class. This is a rare exception. You should use `await` everywhere else!

When you start the application next, the `admin@todo.local` account will be created and assigned the Administrator role. Try logging in with this account, and navigating to `http://localhost:5000/ManageUsers`. You'll see a list of all users registered for the application.

> As an extra challenge, try adding more administration features to this page. For example, you could add a button that gives an administrator the ability to delete a user account.

Check for authorization in a view

The `[Authorize]` attribute makes it easy to perform an authorization check in a controller or action method, but what if you need to check authorization in a view? For example, it would be nice to display a "Manage users" link in the navigation bar if the logged-in user is an administrator.

You can inject the `UserManager` directly into a view to do these types of authorization checks. To keep your views clean and organized, create a new partial view that will add an item to the navbar in the layout:

Views/Shared/_AdminActionsPartial.cshtml

```
@using Microsoft.AspNetCore.Identity
@using AspNetCoreTodo.Models

@inject SignInManager<ApplicationUser> signInManager
@inject UserManager<ApplicationUser> userManager

@if (signInManager.IsSignedIn(User))
{
    var currentUser = await userManager.GetUserAsync
(User);

    var isAdmin = currentUser != null
        && await userManager.IsInRoleAsync(
            currentUser,
            Constants.AdministratorRole);

    if (isAdmin)
    {
        <ul class="nav navbar-nav navbar-right">
            <li>
                <a asp-controller="ManageUsers"
                   asp-action="Index">
                   Manage Users
                </a>
            </li>
        </ul>
    }
}
```

> It's conventional to name shared partial views starting with an `_` underscore, but it's not required.

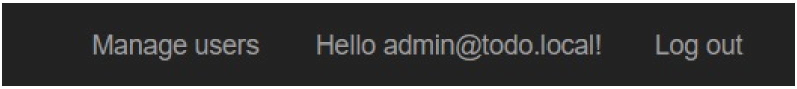
This partial view first uses the `SignInManager` to quickly determine whether the user is logged in. If they aren't, the rest of the view code can be skipped. If there **is** a logged-in user, the `UserManager` is used to look up their details and perform an authorization check with `IsInRoleAsync()`. If all checks succeed and the user is an adminstrator, a **Manage users** link is added to the navbar.

To include this partial in the main layout, edit `_Layout.cshtml` and add it in the navbar section:

Views/Shared/_Layout.cshtml

```
<div class="navbar-collapse collapse">
    <ul class="nav navbar-nav">
        <!-- existing code here -->
    </ul>
    @await Html.PartialAsync("_LoginPartial")
    @await Html.PartialAsync("_AdminActionsPartial")
</div>
```

When you log in with an administrator account, you'll now see a new item on the top right:



More resources

ASP.NET Core Identity helps you add security and identity features like login and registration to your application. The `dotnet new` templates give you pre-built views and controllers that handle these common scenarios so you can get up and running quickly.

There's much more that ASP.NET Core Identity can do, such as password reset and social login. The documentation available at http://docs.asp.net is a fantastic resource for learning how to add these features.

Alternatives to ASP.NET Core Identity

ASP.NET Core Identity isn't the only way to add identity functionality. Another approach is to use a cloud-hosted identity service like Azure Active Directory B2C or Okta to handle identity for your application. You can think of these options as part of a progression:

- **Do-it-yourself security**: Not recommended, unless you are a security expert!
- **ASP.NET Core Identity**: You get a lot of code for free with the templates, which makes it pretty easy to get started. You'll still need to write some code for more

advanced scenarios, and maintain a database to store user information.

- **Cloud-hosted identity services**. The service handles both simple and advanced scenarios (multi-factor authentication, account recovery, federation), and significantly reduces the amount of code you need to write and maintain in your application. Plus, sensitive user data isn't stored in your own database.

For this project, ASP.NET Core Identity is a great fit. For more complex projects, I'd recommend doing some research and experimenting with both options to understand which is best for your use case.

Automated testing

Writing tests is an important part of building any application. Testing your code helps you find and avoid bugs, and makes it easier to refactor your code later without breaking functionality or introducing new problems.

In this chapter you'll learn how to write both **unit tests** and **integration tests** that exercise your ASP.NET Core application. Unit tests are small tests that make sure a single method or chunk of logic works properly. Integration tests (sometimes called **functional** tests) are larger tests that simulate real-world scenarios and test multiple layers or parts of your application.

Unit testing

Unit tests are small, short tests that check the behavior of a single method or class. When the code you're testing relies on other methods or classes, unit tests rely on **mocking** those other classes so that the test only focuses on one thing at a time.

For example, the `TodoController` class has two dependencies: an `ITodoItemService` and the `UserManager`. The `TodoItemService`, in turn, depends on the `ApplicationDbContext`. (The idea that you can draw a line from `TodoController` > `TodoItemService` > `ApplicationDbContext` is called a **dependency graph**).

When the application runs normally, the ASP.NET Core service container and dependency injection system injects each of those objects into the dependency graph when the `TodoController` or the `TodoItemService` is created.

When you write a unit test, on the other hand, you have to handle the dependency graph yourself. It's typical to provide test-only or "mocked" versions of those dependencies. This means you can isolate just the logic in the class or method you are testing. (This is important! If you're testing a service, you don't want to **also** be accidentally writing to your database.)

Create a test project

It's a best practice to create a separate project for your tests, so they are kept separate from your application code. The new test project should live in a directory that's next to (not inside) your main project's directory.

If you're currently in your project directory, `cd` up one level. (This root directory will also be called `AspNetCoreTodo`). Then use this command to scaffold a new test project:

```
dotnet new xunit -o AspNetCoreTodo.UnitTests
```

xUnit.NET is a popular test framework for .NET code that can be used to write both unit and integration tests. Like everything else, it's a set of NuGet packages that can be installed in any project. The `dotnet new xunit` template already includes everything you need.

Your directory structure should now look like this:

```
AspNetCoreTodo/
    AspNetCoreTodo/
        AspNetCoreTodo.csproj
        Controllers/
        (etc...)

    AspNetCoreTodo.UnitTests/
        AspNetCoreTodo.UnitTests.csproj
```

Since the test project will use the classes defined in your main project, you'll need to add a reference to the `AspNetCoreTodo` project:

```
dotnet add reference ../AspNetCoreTodo/AspNetCoreTodo.csproj
```

Delete the `UnitTest1.cs` file that's automatically created. You're ready to write your first test.

> If you're using Visual Studio Code, you may need to close and reopen the Visual Studio Code window to get code completion working in the new project.

Write a service test

Take a look at the logic in the `AddItemAsync()` method of the `TodoItemService`:

```
public async Task<bool> AddItemAsync(
    TodoItem newItem, ApplicationUser user)
{
    newItem.Id = Guid.NewGuid();
    newItem.IsDone = false;
    newItem.DueAt = DateTimeOffset.Now.AddDays(3);
    newItem.UserId = user.Id;

    _context.Items.Add(newItem);

    var saveResult = await _context.SaveChangesAsync
();
    return saveResult == 1;
}
```

This method makes a number of decisions or assumptions about the new item (in other words, performs business logic on the new item) before it actually saves it to the database:

- The `UserId` property should be set to the user's ID
- New items should always be incomplete (`IsDone = false`)
- The title of the new item should be copied from `newItem.Title`
- New items should always be due 3 days from now

Imagine if you or someone else refactored the `AddItemAsync()` method and forgot about part of this business logic. The behavior of your application could change without

you realizing it! You can prevent this by writing a test that double-checks that this business logic hasn't changed (even if the method's internal implementation changes).

> It might seem unlikely now that you could introduce a change in business logic without realizing it, but it becomes much harder to keep track of decisions and assumptions in a large, complex project. The larger your project is, the more important it is to have automated checks that make sure nothing has changed!

To write a unit test that will verify the logic in the `TodoItemService`, create a new class in your test project:

AspNetCoreTodo.UnitTests/TodoItemServiceShould.cs

```
using System;
using System.Threading.Tasks;
using AspNetCoreTodo.Data;
using AspNetCoreTodo.Models;
using AspNetCoreTodo.Services;
using Microsoft.EntityFrameworkCore;
using Xunit;

namespace AspNetCoreTodo.UnitTests
{
    public class TodoItemServiceShould
    {
        [Fact]
        public async Task AddNewItemAsIncompleteWith
DueDate()
        {
            // ...
        }
    }
}
```

> There are many different ways of naming and organizing tests, all with different pros and cons. I like postfixing my test classes with `Should` to create a readable sentence with the test method name, but feel free to use your own style!

The `[Fact]` attribute comes from the xUnit.NET package, and it marks this method as a test method.

The `TodoItemService` requires an `ApplicationDbContext`, which is normally connected to your database. You won't want to use that for tests. Instead, you can use Entity Framework Core's in-memory database provider in your test code. Since the entire database exists in memory, it's wiped out every time the test is restarted. And, since it's a proper Entity Framework Core provider, the `TodoItemService` won't know the difference!

Use a `DbContextOptionsBuilder` to configure the in-memory database provider, and then make a call to `AddItemAsync()`:

```
var options = new DbContextOptionsBuilder<Applicatio
nDbContext>()
    .UseInMemoryDatabase(databaseName: "Test_AddNewI
tem").Options;

// Set up a context (connection to the "DB") for wri
ting
using (var context = new ApplicationDbContext(option
s))
{
    var service = new TodoItemService(context);

    var fakeUser = new ApplicationUser
    {
        Id = "fake-000",
        UserName = "fake@example.com"
    };

    await service.AddItemAsync(new TodoItem
    {
        Title = "Testing?"
    }, fakeUser);
}
```

The last line creates a new to-do item called `Testing?`, and tells the service to save it to the (in-memory) database.

To verify that the business logic ran correctly, write some more code below the existing `using` block:

```
// Use a separate context to read data back from the
 "DB"
using (var context = new ApplicationDbContext(option
s))
{
    var itemsInDatabase = await context
        .Items.CountAsync();
    Assert.Equal(1, itemsInDatabase);

    var item = await context.Items.FirstAsync();
    Assert.Equal("Testing?", item.Title);
    Assert.Equal(false, item.IsDone);

    // Item should be due 3 days from now (give or t
ake a second)
    var difference = DateTimeOffset.Now.AddDays(3) -
 item.DueAt;
    Assert.True(difference < TimeSpan.FromSeconds(1)
);
}
```

The first assertion is a sanity check: there should never be more than one item saved to the in-memory database. Assuming that's true, the test retrieves the saved item with `FirstAsync` and then asserts that the properties are set to the expected values.

> Both unit and integration tests typically follow the AAA (Arrange-Act-Assert) pattern: objects and data are set up first, then some action is performed, and finally the test checks (asserts) that the expected behavior occurred.

Asserting a datetime value is a little tricky, since comparing two dates for equality will fail if even the millisecond components are different. Instead, the test checks that the `DueAt` value is less than a second away from the expected value.

Run the test

On the terminal, run this command (make sure you're still in the `AspNetCoreTodo.UnitTests` directory):

```
dotnet test
```

The `test` command scans the current project for tests (marked with `[Fact]` attributes in this case), and runs all the tests it finds. You'll see output similar to:

```
Starting test execution, please wait...
 Discovering: AspNetCoreTodo.UnitTests
 Discovered:  AspNetCoreTodo.UnitTests
 Starting:    AspNetCoreTodo.UnitTests
 Finished:    AspNetCoreTodo.UnitTests

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.9074 Seconds
```

You now have one test providing test coverage of the `TodoItemService`. As an extra challenge, try writing unit tests that ensure:

- The `MarkDoneAsync()` method returns false if it's passed an ID that doesn't exist
- The `MarkDoneAsync()` method returns true when it makes a valid item as complete
- The `GetIncompleteItemsAsync()` method returns only the items owned by a particular user

Integration testing

Compared to unit tests, integration tests are much larger in scope. exercise the whole application stack. Instead of isolating one class or method, integration tests ensure that all of the components of your application are working together properly: routing, controllers, services, database code, and so on.

Integration tests are slower and more involved than unit tests, so it's common for a project to have lots of small unit tests but only a handful of integration tests.

In order to test the whole stack (including controller routing), integration tests typically make HTTP calls to your application just like a web browser would.

To perform an integration test, you could start your application and manually make requests to http://localhost:5000. However, ASP.NET Core provides a better alternative: the `TestServer` class. This class can host your application for the duration of the test, and then stop it automatically when the test is complete.

Create a test project

If you're currently in your project directory, `cd` up one level to the root `AspNetCoreTodo` directory. Use this command to scaffold a new test project:

```
dotnet new xunit -o AspNetCoreTodo.IntegrationTests
```

Your directory structure should now look like this:

```
AspNetCoreTodo/
    AspNetCoreTodo/
        AspNetCoreTodo.csproj
        Controllers/
        (etc...)

    AspNetCoreTodo.UnitTests/
        AspNetCoreTodo.UnitTests.csproj

    AspNetCoreTodo.IntegrationTests/
        AspNetCoreTodo.IntegrationTests.csproj
```

> If you prefer, you can keep your unit tests and integration tests in the same project. For large projects, it's common to split them up so it's easy to run them separately.

Since the test project will use the classes defined in your main project, you'll need to add a reference to the main project:

```
dotnet add reference ../AspNetCoreTodo/AspNetCoreTod
o.csproj
```

You'll also need to add the `Microsoft.AspNetCore.TestHost` NuGet package:

```
dotnet add package Microsoft.AspNetCore.TestHost
```

Delete the `UnitTest1.cs` file that's created by `dotnet new`. You're ready to write an integration test.

Write an integration test

There are a few things that need to be configured on the test server before each test. Instead of cluttering the test with this setup code, you can keep this setup in a separate class. Create a new class called `TestFixture`:

AspNetCoreTodo.IntegrationTests/TestFixture.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Net.Http;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.TestHost;
using Microsoft.Extensions.Configuration;

namespace AspNetCoreTodo.IntegrationTests
{
    public class TestFixture : IDisposable
    {
        private readonly TestServer _server;
```

```
        public HttpClient Client { get; }

        public TestFixture()
        {
            var builder = new WebHostBuilder()
                .UseStartup<AspNetCoreTodo.Startup>(
)
                .ConfigureAppConfiguration((context,
 config) =>
                {
                    config.SetBasePath(Path.Combine(
                        Directory.GetCurrentDirector
y(),
                        "..\\..\\..\\..\\AspNetCoreT
odo"));

                    config.AddJsonFile("appsettings.
json");
                });

            _server = new TestServer(builder);

            Client = _server.CreateClient();
            Client.BaseAddress = new Uri("http://loc
alhost:8888");
        }

        public void Dispose()
        {
            Client.Dispose();
            _server.Dispose();
        }
    }
}
```

This class takes care of setting up a `TestServer`, and will help keep the tests themselves clean and tidy.

Now you're (really) ready to write an integration test. Create a new class called `TodoRouteShould`:

AspNetCoreTodo.IntegrationTests/TodoRouteShould.cs

```
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using Xunit;

namespace AspNetCoreTodo.IntegrationTests
{
    public class TodoRouteShould : IClassFixture<Tes
tFixture>
    {
        private readonly HttpClient _client;

        public TodoRouteShould(TestFixture fixture)
        {
            _client = fixture.Client;
        }

        [Fact]
        public async Task ChallengeAnonymousUser()
        {
            // Arrange
            var request = new HttpRequestMessage(
                HttpMethod.Get, "/todo");
```

```
            // Act: request the /todo route
            var response = await _client.SendAsync(r
equest);

            // Assert: the user is sent to the login
 page
            Assert.Equal(
                HttpStatusCode.Redirect,
                response.StatusCode);

            Assert.Equal(
                "http://localhost:8888/Account" +
                "/Login?ReturnUrl=%2Ftodo",
                response.Headers.Location.ToString()
);
        }
    }
}
```

This test makes an anonymous (not-logged-in) request to the `/todo` route and verifies that the browser is redirected to the login page.

This scenario is a good candidate for an integration test, because it involves multiple components of the application: the routing system, the controller, the fact that the controller is marked with `[Authorize]`, and so on. It's also a good test because it ensures you won't ever accidentally remove the `[Authorize]` attribute and make the to-do view accessible to everyone.

Run the test

Run the test in the terminal with `dotnet test`. If everything's working right, you'll see a success message:

```
Starting test execution, please wait...
 Discovering: AspNetCoreTodo.IntegrationTests
 Discovered:  AspNetCoreTodo.IntegrationTests
 Starting:    AspNetCoreTodo.IntegrationTests
 Finished:    AspNetCoreTodo.IntegrationTests

Total tests: 1. Passed: 1. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 2.0588 Seconds
```

Wrap up

Testing is a broad topic, and there's much more to learn. This chapter doesn't touch on UI testing or testing frontend (JavaScript) code, which probably deserve entire books of their own. You should, however, have the skills and base knowledge you need to learn more about testing and to practice writing tests for your own applications.

The ASP.NET Core documentation (https://docs.asp.net) and Stack Overflow are great resources for learning more and finding answers when you get stuck.

Deploy the application

You've come a long way, but you're not quite done yet. Once you've created a great application, you need to share it with the world!

Because ASP.NET Core applications can run on Windows, Mac, or Linux, there are a number of different ways you can deploy your application. In this chapter, I'll show you the most common (and easiest) ways to go live.

Deployment options

ASP.NET Core applications are typically deployed to one of these environments:

- **A Docker host**. Any machine capable of hosting Docker containers can be used to host an ASP.NET Core application. Creating a Docker image is a very quick way to get your application deployed, especially if you're familiar with Docker. (If you're not, don't worry! I'll cover the steps later.)

- **Azure**. Microsoft Azure has native support for ASP.NET Core applications. If you have an Azure subscription, you just need to create a Web App and upload your project files. I'll cover how to do this with the Azure CLI in the next section.
- **Linux (with Nginx)**. If you don't want to go the Docker route, you can still host your application on any Linux server (this includes Amazon EC2 and DigitalOcean virtual machines). It's typical to pair ASP.NET Core with the Nginx reverse proxy. (More about Nginx below.)
- **Windows**. You can use the IIS web server on Windows to host ASP.NET Core applications. It's usually easier (and cheaper) to just deploy to Azure, but if you prefer managing Windows servers yourself, it'll work just fine.

Kestrel and reverse proxies

> If you don't care about the guts of hosting ASP.NET Core applications and just want the step-by-step instructions, feel free to skip to one of the next two sections.

ASP.NET Core includes a fast, lightweight web server called Kestrel. It's the server you've been using every time you ran `dotnet run` and browsed to `http://localhost:5000`. When you deploy your application to a production environment, it'll

still use Kestrel behind the scenes. However, it's recommended that you put a reverse proxy in front of Kestrel, because Kestrel doesn't yet have load balancing and other features that more mature web servers have.

On Linux (and in Docker containers), you can use Nginx or the Apache web server to receive incoming requests from the internet and route them to your application hosted with Kestrel. If you're on Windows, IIS does the same thing.

If you're using Azure to host your application, this is all done for you automatically. I'll cover setting up Nginx as a reverse proxy in the Docker section.

Deploy to Azure

Deploying your ASP.NET Core application to Azure only takes a few steps. You can do it through the Azure web portal, or on the command line using the Azure CLI. I'll cover the latter.

What you'll need

- Git (use `git --version` to make sure it's installed)
- The Azure CLI (follow the install instructions at https://github.com/Azure/azure-cli)
- An Azure subscription (the free subscription is fine)
- A deployment configuration file in your project root

Create a deployment configuration file

Since there are multiple projects in your directory structure (the web application, and two test projects), Azure won't know which one to publish. To fix this, create a file called `.deployment` at the very top of your directory structure:

.deployment

```
[config]
project = AspNetCoreTodo/AspNetCoreTodo.csproj
```

Make sure you save the file as `.deployment` with no other parts to the name. (On Windows, you may need to put quotes around the filename, like `".deployment"`, to prevent a `.txt` extension from being added.)

If you `ls` or `dir` in your top-level directory, you should see these items:

```
.deployment
AspNetCoreTodo
AspNetCoreTodo.IntegrationTests
AspNetCoreTodo.UnitTests
```

Set up the Azure resources

If you just installed the Azure CLI for the first time, run

```
az login
```

and follow the prompts to log in on your machine. Then, create a new Resource Group for this application:

```
az group create -l westus -n AspNetCoreTodoGroup
```

This creates a Resource Group in the West US region. If you're located far away from the western US, use `az account list-locations` to get a list of locations and find one closer to you.

Next, create an App Service plan in the group you just created:

```
az appservice plan create -g AspNetCoreTodoGroup -n
AspNetCoreTodoPlan --sku F1
```

> F1 is the free app plan. If you want to use a custom domain name with your app, use the D1 ($10/month) plan or higher.

Now create a Web App in the App Service plan:

```
az webapp create -g AspNetCoreTodoGroup -p AspNetCor
eTodoPlan -n MyTodoApp
```

The name of the app (`MyTodoApp` above) must be globally unique in Azure. Once the app is created, it will have a default URL in the format: http://mytodoapp.azurewebsites.net

Deploy your project files to Azure

You can use Git to push your application files up to the Azure Web App. If your local directory isn't already tracked as a Git repo, run these commands to set it up:

```
git init
git add .
git commit -m "First commit!"
```

Next, create an Azure username and password for deployment:

```
az webapp deployment user set --user-name nate
```

Follow the instructions to create a password. Then use `config-local-git` to spit out a Git URL:

```
az webapp deployment source config-local-git -g AspN
etCoreTodoGroup -n MyTodoApp --out tsv

https://nate@mytodoapp.scm.azurewebsites.net/MyTodoA
pp.git
```

Copy the URL to the clipboard, and use it to add a Git remote to your local repository:

```
git remote add azure <paste>
```

You only need to do these steps once. Now, whenever you want to push your application files to Azure, check them in with Git and run

```
git push azure master
```

You'll see a stream of log messages as the application is deployed to Azure.

When it's complete, browse to http://yourappname.azurewebsites.net to check out the app!

Deploy with Docker

If you aren't using a platform like Azure, containerization technologies like Docker can make it easy to deploy web applications to your own servers. Instead of spending time configuring a server with the dependencies it needs to run your app, copying files, and restarting processes, you can simply create a Docker image that describes everything your app needs to run, and spin it up as a container on any Docker host.

Docker can make scaling your app across multiple servers easier, too. Once you have an image, using it to create 1 container is the same process as creating 100 containers.

Before you start, you need the Docker CLI installed on your development machine. Search for "get docker for (mac/windows/linux)" and follow the instructions on the official Docker website. You can verify that it's installed correctly with

```
docker version
```

Add a Dockerfile

The first thing you'll need is a Dockerfile, which is like a recipe that tells Docker what your application needs to build and run.

Create a file called `Dockerfile` (no extension) in the root, top-level `AspNetCoreTodo` folder. Open it in your favorite editor. Write the following line:

```
FROM microsoft/dotnet:2.0-sdk AS build
```

This tells Docker to use the `microsoft/dotnet:2.0-sdk` image as a starting point. This image is published by Microsoft and contains the tools and dependencies you need to execute `dotnet build` and compile your application. By using this pre-built image as a starting point, Docker can optimize the image produced for your app and keep it small.

Next, add this line:

```
COPY AspNetCoreTodo/*.csproj ./app/AspNetCoreTodo/
```

The `COPY` command copies the `.csproj` project file into the image at the path `/app/AspNetCoreTodo/`. Note that none of the actual code (`.cs` files) have been copied into the image yet. You'll see why in a minute.

```
WORKDIR /app/AspNetCoreTodo
RUN dotnet restore
```

`WORKDIR` is the Docker equivalent of `cd`. This means any commands executed next will run from inside the `/app/AspNetCoreTodo` directory that the `COPY` command created in the last step.

Running the `dotnet restore` command restores the NuGet packages that the application needs, defined in the `.csproj` file. By restoring packages inside the image **before** adding the rest of the code, Docker is able to cache the restored packages. Then, if you make code changes (but don't change the packages defined in the project file), rebuilding the Docker image will be super fast.

Now it's time to copy the rest of the code and compile the application:

```
COPY AspNetCoreTodo/. ./AspNetCoreTodo/
RUN dotnet publish -o out /p:PublishWithAspNetCoreTa
rgetManifest="false"
```

The `dotnet publish` command compiles the project, and the `-o out` flag puts the compiled files in a directory called `out`.

These compiled files will be used to run the application with the final few commands:

```
FROM microsoft/dotnet:2.0-runtime AS runtime
ENV ASPNETCORE_URLS http://+:80
WORKDIR /app
COPY --from=build /app/AspNetCoreTodo/out ./
ENTRYPOINT ["dotnet", "AspNetCoreTodo.dll"]
```

The `FROM` command is used again to select a smaller image that only has the dependencies needed to run the application. The `ENV` command is used to set environment variables in the container, and the `ASPNETCORE_URLS` environment variable tells ASP.NET Core which network interface and port it should bind to (in this case, port 80).

The `ENTRYPOINT` command lets Docker know that the container should be started as an executable by running `dotnet AspNetCoreTodo.dll`. This tells `dotnet` to start up your application from the compiled file created by `dotnet publish` earlier. (When you do `dotnet run` during development, you're accomplishing the same thing in one step.)

The full Dockerfile looks like this:

Dockerfile

```
FROM microsoft/dotnet:2.0-sdk AS build
COPY AspNetCoreTodo/*.csproj ./app/AspNetCoreTodo/
WORKDIR /app/AspNetCoreTodo
RUN dotnet restore

COPY AspNetCoreTodo/. ./
RUN dotnet publish -o out /p:PublishWithAspNetCoreTa
rgetManifest="false"

FROM microsoft/dotnet:2.0-runtime AS runtime
ENV ASPNETCORE_URLS http://+:80
WORKDIR /app
COPY --from=build /app/AspNetCoreTodo/out ./
ENTRYPOINT ["dotnet", "AspNetCoreTodo.dll"]
```

Create an image

Make sure the Dockerfile is saved, and then use `docker build` to create an image:

```
docker build -t aspnetcoretodo .
```

Don't miss the trailing period! That tells Docker to look for a Dockerfile in the current directory.

Once the image is created, you can run `docker images` to to list all the images available on your local machine. To test it out in a container, run

```
docker run --name aspnetcoretodo_sample --rm -it -p
8080:80 aspnetcoretodo
```

The `-it` flag tells Docker to run the container in interactive mode (outputting to the terminal, as opposed to running in the background). When you want to stop the container, press Control-C.

Remember the `ASPNETCORE_URLS` variable that told ASP.NET Core to listen on port 80? The `-p 8080:80` option tells Docker to map port 8080 on *your* machine to the *container's* port 80.

Open up your browser and navigate to http://localhost:8080 to see the application running in the container!

Set up Nginx

At the beginning of this chapter, I mentioned that you should use a reverse proxy like Nginx to proxy requests to Kestrel. You can use Docker for this, too.

The overall architecture will consist of two containers: an Nginx container listening on port 80, forwarding requests to the container you just built that hosts your application with Kestrel.

The Nginx container needs its own Dockerfile. To keep it from conflicting with the Dockerfile you just created, make a new directory in the web application root:

```
mkdir nginx
```

Create a new Dockerfile and add these lines:

nginx/Dockerfile

```
FROM nginx
COPY nginx.conf /etc/nginx/nginx.conf
```

Next, create an `nginx.conf` file:

nginx/nginx.conf

```
events { worker_connections 1024; }

http {
    server {
        listen 80;
        location / {
          proxy_pass http://kestrel:80;
          proxy_http_version 1.1;
          proxy_set_header Upgrade $http_upgrade;
          proxy_set_header Connection 'keep-alive';
          proxy_set_header Host $host;
          proxy_cache_bypass $http_upgrade;
        }
    }
}
```

This configuration file tells Nginx to proxy incoming requests to `http://kestrel:80`. (You'll see why `kestrel` works as a hostname in a moment.)

> When you make deploy your application to a production environment, you should add the `server_name` directive and validate and restrict the host header to known good values. For more information, see:
>
> https://github.com/aspnet/Announcements/issues/295

Set up Docker Compose

There's one more file to create. Up in the root directory, create `docker-compose.yml`:

docker-compose.yml

```
nginx:
    build: ./nginx
    links:
        - kestrel:kestrel
    ports:
        - "80:80"
kestrel:
    build: .
    ports:
        - "80"
```

Docker Compose is a tool that helps you create and run multi-container applications. This configuration file defines two containers: `nginx` from the `./nginx/Dockerfile` recipe, and `kestrel` from the `./Dockerfile` recipe. The containers are explicitly linked together so they can communicate.

You can try spinning up the entire multi-container application by running:

```
docker-compose up
```

Try opening a browser and navigating to http://localhost (port 80, not 8080!). Nginx is listening on port 80 (the default HTTP port) and proxying requests to your ASP.NET Core application hosted by Kestrel.

Set up a Docker server

Specific setup instructions are outside the scope of this book, but any modern flavor of Linux (like Ubuntu) can be used to set up a Docker host. For example, you could create a virtual machine with Amazon EC2, and install the Docker service. You can search for "amazon ec2 set up docker" (for example) for instructions.

I like using DigitalOcean because they've made it really easy to get started. DigitalOcean has both a pre-built Docker virtual machine, and in-depth tutorials for getting Docker up and running (search for "digitalocean docker").

Conclusion

Thanks for making it to the end of the Little ASP.NET Core Book! If this book was helpful (or not), I'd love to hear your thoughts. Send me your comments via Twitter: https://twitter.com/nbarbettini

How to learn more

There's a lot more that ASP.NET Core can do that couldn't fit in this short book, including

- Building RESTful APIs and microservices
- Using ASP.NET Core with single-page apps like Angular and React
- Razor Pages
- Bundling and minifying static assets
- WebSockets and SignalR

There are a number of ways you can learn more:

- **The ASP.NET Core documentation.** The official ASP.NET Core documentation at http://docs.asp.net contains a number of in-depth tutorials covering many of these topics. I'd highly recommend it!

- **ASP.NET Core in Action.** This book by Andrew Lock is a comprehensive, deep dive into ASP.NET Core. You can get it from Amazon or a local bookstore.
- **Courses on LinkedIn Learning and Pluralsight.** If you learn best from videos, there are fantastic courses available on Pluralsight and LinkedIn Learning (including some by yours truly). If you don't have an account and need a coupon, send me an email: nate@barbettini.com.
- **Nate's blog.** I also write about ASP.NET Core and more on my blog at https://www.recaffeinate.co.

Happy coding!

About the author

Hey, I'm Nate! I wrote the Little ASP.NET Core Book in a long, caffeine-fueled weekend because I love the .NET community and wanted to give back in my own little way. I hope it helped you learn something new!

You can stay in touch with me on Twitter (@nbarbettini) or on my blog (https://www.recaffeinate.co). You can also reach me via email at nate@barbettini.com.

Special thanks

To Jennifer, who always supports my crazy ideas.

To the following contributors who improved the Little ASP.NET Core Book:

- 0xNF
- Matt Welke (welke)
- Raman Zhylich (zhilich)

To these amazing polyglot programmers who translated the Little ASP.NET Core Book:

- sahinyanlik (Turkish)
- windsting, yuyi (Simplified Chinese)

Changelog

The full, detailed changelog is always available here:

https://github.com/nbarbettini/little-aspnetcore-book/releases

1.1.1 (2018-06-11): Fixed typos found by readers.

1.1.0 (2018-05-03): Significantly reworked the *Add more features* chapter to use MVC thorough the whole stack and remove the AJAX pattern. Removed Facebook login to simplify the security chapter and streamline testing and deployment.

Updated the Docker instructions to reflect the latest best practices. Fixed typos and added suggestions from readers. The book also sports a new, improved cover design!

1.0.4 (2018-01-15): Added explanation of service container lifecycles, clarified server ports and the -o flag, and removed semicolons after Razor directives. Corrected Chinese translation author credit. Fixed other small typos and issues noticed by readers.

1.0.3 (2017-11-13): Typo fixes and small improvements suggested by readers.

1.0.2 (2017-10-20): More bug fixes and small improvements. Added link to translations.

1.0.1 (2017-09-23): Bug fixes and small improvements.

1.0.0 (2017-09-18): Initial release.